Informatik
Hauptcampus

HOCHSCHULE TRIER

Zielgerichtetes Reasoning in der Prädikatenlogik durch Lernen aus Beweisen mit Hilfe großer Sprachmodelle

Goal-Directed Reasoning in First-Order Logic through Learning from Proofs Using Large Language Models

Fabian Koßmann

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Claudia Schon

Trier, 31.05.2024

Vorwort

Die vorliegende Bachelorarbeit entstand im Rahmen meines Studiums der Informatik an der Hochschule Trier, mit dem Abschluss Bachelor of Science.
Die Wahl des Themas „Zielgerichtetes Reasoning in der Prädikatenlogik durch Lernen aus Beweisen mit Hilfe großer Sprachmodelle“ ergab sich aus meinem wachsenden Interesse in den Bereichen Logik und künstliche Intelligenz, das sich im Laufe meines Studiums entwickelt hat.
Diese Begeisterung hat mich motiviert, meine Bachelorarbeit in diesem spannenden und zukunftsweisenden Feld zu schreiben.
An dieser Stelle möchte ich mich herzlich bei allen bedanken, die mich auf diesem Weg unterstützt haben.
Mein besonderer Dank gilt meiner Professorin, Prof. Dr. Claudia Schon, die sich regelmäßig mit mir getroffen, viel Zeit investiert und große Geduld bewiesen hat. Ihre wertvollen Vorschläge und Ideen sowie die sorgfältige Korrektur meiner Arbeit waren von unschätzbarem Wert.
Ebenso danke ich der Hochschule Trier und allen Professoren, die mich auf meinem Weg zum Bachelorabschluss begleitet haben.
Ein großer Dank gebührt auch meinen Eltern und Geschwistern, die mich während meines gesamten Bachelorstudiums unterstützt haben. Ihre beständige Ermutigung und Unterstützung waren eine große Hilfe für mich. Besonders möchte ich auch meiner Oma danken, die mich während meines gesamten Studiums finanziell unterstützt hat.
Die Ziele, die ich mit dieser Arbeit verfolgt habe, waren vielfältig. Ich wollte mein Wissen im Bereich Reasoning und neuronale Netze erweitern, ein vertieftes Verständnis von Logik erlangen und die Erfahrung sammeln, ein komplexes Problem systematisch zu bearbeiten. Die Erstellung dieser Arbeit war eine herausfordernde, aber auch äußerst bereichernde Erfahrung, die meinen Horizont sowohl fachlich als auch persönlich erweitert hat. Ich hoffe, dass die gewonnenen Erkenntnisse einen wertvollen Beitrag zur Forschung im Bereich Künstliche Intelligenz leisten können.

Kurzfassung

Diese Bachelorarbeit befasst sich mit der Verbesserung der Beweisführung durch das Lernen, wie gut prädikatenlogische Symbole zu einer Beweisaufgabe passen, ausgehend von erfolgreich gefundenen Beweisen. Der Lösungsansatz umfasst die Verwendung des Eprovers zur Beweisführung, das Fine-tuning eines großen Sprachmodells und das Scoring, wie gut Symbole aus bereits gefundenen Beweisen zu neuen Beweisaufgaben passen. Untersucht werden die Anzahl der verarbeiteten Klauseln, die gefundenen Beweise und die Entdeckung neuer, zuvor nicht gefundener Beweise durch die Bewertung der Symbolrelevanz mittels des Sprachmodells, das dem Eprover gezielte Hinweise gibt. Die Ergebnisse zeigen, dass der Beweisprozess durch das Lernen mittels eines großen Sprachmodells, wie gut prädikatenlogische Symbole zu einer Beweisaufgabe passen, verbessert werden kann. Insbesondere konnte die Anzahl der verarbeiteten Klauseln deutlich reduziert werden, jedoch ging dies mit einer geringeren Anzahl gefundener Beweise einher. Trotzdem konnten neue, zuvor nicht gefundene Beweise entdeckt werden.

Abstract

This thesis examines the improvement of proof guidance by learning how well predicate logic symbols fit a proof task, based on successfully found proofs. The solution approach involves using the Eprover to solve proofs, fine-tuning a large language model, and scoring how well symbols from previously found proofs match new proof tasks. The work investigates the number of processed clauses, the proofs found, and the discovery of new, previously unfound proofs by evaluating the symbol relevance with the language model, which provides targeted guidance to the Eprover. The results indicate that the proof process can be enhanced by learning with a language model how well predicate logic symbols fit a proof task. Specifically, the number of processed clauses was significantly reduced, although this led to a decrease in the number of proofs found. Nevertheless, new, previously unfound proofs were successfully discovered.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

1.1 Hintergrund und Relevanz der Thematik

Automatisches Beweisen stellt ein bedeutendes Teilgebiet der Künstlichen Intelligenz dar.
Ein weiteres wichtiges Teilgebiet der Künstlichen Intelligenz sind Sprachmodelle. In jüngster Zeit stehen insbesondere große Transformer-Sprachmodelle wie ChatGPT im Fokus der Aufmerksamkeit. Jedoch geht mit ihrer Verwendung ein Nachteil einher: Die Herangehensweise ist rein statistisch und trainingsbasiert, was die Nachvollziehbarkeit der Ergebnisse erschwert.
Im Gegensatz dazu zeichnet sich das Beweisen durch eine klare Abfolge von logischen Schritten aus, um zu einem Beweis zu gelangen, bei der jeder Schritt nachvollziehbar ist. Hier liegt der Fokus auf dem Beweis und darauf, dass alle Beweisschritte nachvollziehbar sind.
Dies ist insbesondere in sicherheitskritischen Bereichen von großer Bedeutung, wie beispielsweise der Softwareverifikation und der IT-Sicherheit, wo die Genauigkeit und Verlässlichkeit der Ergebnisse entscheidend ist.
Ein Hindernis bei der Anwendung von automatischen Beweisen in der Praxis liegt in der Notwendigkeit einer formalen Sprache, wie der Prädikatenlogik, zur Formulierung wohldefinierter und widerspruchsfreier Beweisziele. Zudem bedarf es einer Wissensbasis in Form von Axiomen sowie Schlussregeln in Form eines Kalküls, um Folgerungen zu ziehen.
Ein weiterer Nachteil liegt in der potenziellen Unentscheidbarkeit bestimmter Logiken wie der Prädikatenlogik, was bedeuten kann, dass ein Beweiser endlos nach einer Lösung sucht.
Beweiser berücksichtigen die Bedeutung von Symbolnamen in der Wissensbasis während des Beweisprozesses normalerweise nicht. In der Regel sind die Symbolnamen jedoch nicht willkürlich gewählt, sondern haben eine Bedeutung, die im Beweisprozess bisher nicht beachtet wird. Natural Language Processing (NLP)-Techniken können dabei helfen, die Bedeutung der Symbolnamen zu analysieren und diese Bedeutung in den Beweisprozess zu integrieren.
Aus diesem Grund besteht eine vielversprechende Lösung darin, die Konzepte zu verbinden, indem beispielsweise Sprachmodelle als Heuristik dienen, um Beweisschritte zu erleichtern und automatische Beweiser zu unterstützen.

1.2 Aktueller Forschungsstand

Automatisches Beweisen ist ein komplexes Thema auf algorithmischer Ebene. Das erste nachgewiesene NP-vollständige Problem war im Jahre 1971 das Erfüllbarkeitsproblem in der Aussagenlogik. Gerade bei einem großen Suchraum und semientscheidbaren Logiken, wie der Prädikatenlogik, sind Heuristiken von großer Bedeutung.
Theorembeweiser arbeiten in der Regel auf Klauselmengen. In jedem Inferenzschritt stehen eine Vielzahl von Klauseln zur Auswahl zur Verfügung. Die Auswahl der Klausel hat einen großen Einfluss auf die Anzahl der Beweisschritte. Die Auswahl spielt deshalb eine entscheidende Rolle für die Effizienz der Beweissuche. Moderne Beweismethoden setzen Heuristiken ein, um die Wahrscheinlichkeit für die Auswahl der richtigen Klausel zu erhöhen.
Aktuelle Heuristiken beruhen hauptsächlich auf der Gewichtung von Klauseln (englisch: Clause Weighting), die dann zur Auswahl herangezogen wird, um mithilfe dieser neue Folgerungen zu ziehen. Bekannte Heuristiken umfassen beispielsweise das Zählen von Symbolen (englisch: Literal Counting), das höhere Gewichten von Klauseln, die viele Symbole enthalten, die ebenfalls im Beweisziel enthalten sind, sowie die Bewertung von Klauseln basierend auf ihrem letzten Verwendungszeitpunkt gemäß dem FIFO- oder LIFO-Prinzip. Eine umfassende Übersicht über aktuelle Heuristiken und einen Vergleich zwischen diesen ist in [SM16] beschrieben.
Die Verwendung von neuronalen Netzen als Heuristik ist ein neueres Forschungsgebiet. Diese Herangehensweise nutzt neuronale Netze, um die Wahrscheinlichkeit für die Auswahl der passenden Klausel vorherzusagen. Dadurch wird die Wahrscheinlichkeit für die Auswahl der richtigen Klausel erhöht. Ein Vorteil dieser Methode besteht darin, dass nicht nur die syntaktischen Aspekte der Symbole, wie sie beim Symbolzählen berücksichtigt werden, sondern auch die Semantik und somit die Bedeutung der Klauseln und ihrer Symbole einbezogen werden können.
Ein interessanter Ansatz wird beispielsweise in der Masterarbeit [McK21] verfolgt, in der ein Einbettungsvektor für Klauseln mithilfe eines neuronalen Netzes erstellt wird, um diesen als Heuristik einzusetzen.
In [FM23] ist ein weiterer interessanter Ansatz beschrieben, bei dem ein neuronales Netz mittels Beweisen trainiert wird, um zukünftige Klauseln besser zu wählen und somit zukünftige Beweise in derselben Domäne zu beschleunigen.

1.3 Zielsetzung der Arbeit

In dieser Arbeit wurde ein großes Sprachmodell benutzt, um aus Beweisen zu lernen. Dieses Modell ist vortrainiert und verfügt bereits über sprachliches Wissen und kennt somit die Ähnlichkeiten von Wörtern. Außerdem sollte es bereits Wissen darüber enthalten, welche Wörter zu einem gemeinsamen Kontext gehören.
Im Rahmen dieser Arbeit wurde die Fragestellung untersucht, ob durch ein Finetuning des Sprachmodells dieses Wissen genutzt werden kann, um bei der Auswahl der „richtigen“ Klausel zu unterstützen.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in drei große Abschnitte:

1. Theoretischer Hintergrund: In diesem Abschnitt werden die Grundlagen der Prädikatenlogik, der Beweistheorie sowie von automatischen Beweismethoden durch Saturation beleuchtet. Zusätzlich werden die Grundlagen neuronaler Netze behandelt.
2. Methodik: In diesem Abschnitt werden die Auswahl einer geeigneten Wissensbasis, die Auswahl eines passenden Beweisers und die Auswahl eines Sprachmodells erläutert. Darüber hinaus wird die Erstellung von Trainings- und Testdaten für das Sprachmodell sowie von Analyse-Daten für die spätere Vergleichsanalyse beschrieben. Diese Analyse untersucht die Nutzung des Beweisers sowohl mit als auch ohne Einsatz des Sprachmodells. Des Weiteren wird das Training (Fine-tuning) des Sprachmodells im Detail erläutert. Zudem erfolgt ein Vergleich der Leistung des Beweisers mit und ohne Sprachmodell, gefolgt von einer Diskussion der Ergebnisse und der Erörterung weiterführender Forschungsansätze.
3. Zusammenfassung und Ausblick: Im letzten Abschnitt werden die Ergebnisse und Herausforderungen, die im Rahmen dieser Arbeit aufgetreten sind, zusammengefasst. Es werden praktische Implikationen dieser Erkenntnisse für die Zukunft erläutert sowie weitere potenzielle Forschungsansätze vorgestellt.

2

Theoretischer Hintergrund

Im folgenden Abschnitt der Bachelorarbeit wird der theoretische Hintergrund beschrieben, der als Grundlage für den späteren Teil der Arbeit dient.

2.1 Prädikatenlogik

Bei diesem Teil der Bachelorarbeit wird die Logik behandelt. Im Abschnitt 2.1.1 wird die Prädikatenlogik und allgemeine Definitionen zur Prädikatenlogik behandelt. Im Abschnitt 2.1.3 werden formale Theorien, Theoreme und Beweise behandelt. Im Teil 2.1.5 werden Herbrand-Theorien erläutert und im Teil 2.1.4 werden Normalformen erläutert. Im letzten Teil 2.1.6 werden dann Theorembeweiser behandelt.

2.1.1 Prädikatenlogik erster Stufe

Im folgenden Abschnitt wird auf die Syntax und Semantik der Prädikatenlogik eingegangen sowie weitere wichtige Definitionen erläutert.

Syntax

Nach [Sch00b] (S. 49-50) werden folgende syntaktischen Elemente für die Prädikatenlogik erster Stufe benötigt.
Ein Prädikatensymbol hat die Form P_i^k, wobei $i \in \mathbb{N}$ und $k \in \mathbb{N}$ sind. Der Index i dient zur Unterscheidung der Symbole, und k ist die Stelligkeit der Relation, die später dem Prädikatensymbol zugeordnet wird. Prädikatensymbole werden in dieser Arbeit mit Kleinbuchstaben benannt.
Ein Funktionssymbol hat die Form f_i^k, wobei $i \in \mathbb{N}$ und $k \in \mathbb{N}$ sind. Der Index i dient zur Unterscheidung der Symbole, und k ist die Stelligkeit der Funktion, die später dem Funktionssymbol zugeordnet wird.
Es gibt auch Symbole f_i^k, wobei k den Wert bzw. die Stelligkeit null hat. Das bedeutet, dass die spätere Funktion für das Symbol kein Argument benötigt. Ein solches Funktionssymbol wird auch Konstante genannt. Funktionssymbole in dieser Arbeit werden mit Kleinbuchstaben benannt.
In der Prädikatenlogik können Variablen verwendet werden. Ein Variablensymbol

hat die Form X_i, wobei $i \in \mathbb{N}$ ist. Die Zahl i dient dazu, die einzelnen Variablensymbole voneinander zu unterscheiden. Variablensymbole werden in dieser Arbeit werden mit Großbuchstaben benannt.
Terme werden induktiv aus Variablensymbolen und Funktionssymbolen gebildet, um später Formeln aus ihnen zu konstruieren.
Laut [Sch00b] (S. 50) werden Terme in der Prädikatenlogik wie folgt definiert:

Definition 2.1 (Terme).

- *Wenn V ein Variablensymbol ist, dann ist V ein Term.*
- *Wenn f ein Funktionssymbol mit der Stelligkeit k ist und $t_1, \ldots, t_k$ Terme sind, dann ist $f(t_1, \ldots, t_k)$ ein Term.*

Mithilfe der definierten Terme können nun induktiv Formeln gebildet werden. Diese sind laut [Sch00b] (S. 50) wie folgt definiert:

Definition 2.2 (Formeln).

- *Wenn P ein Prädikatensymbol mit der Stelligkeit k ist und $t_1, \ldots, t_k$ Terme sind, dann ist $P(t_1, \ldots, t_k)$ eine Formel. Eine solche Formel nennt man auch Atom.*
- *Wenn F und G Formeln sind und X eine Variable ist, dann sind auch folgendes Formeln: $\neg F$, $F \wedge G$, $F \vee G$, $\forall X F$, $\exists X F$.*

Wenn F eine Formel ist, die in einer anderen Formel G enthalten ist, wird F als Teilformel von G bezeichnet.
Die Symbole $\neg$, $\wedge$, $\vee$, $\forall$, $\exists$ werden auch Operatorsymbole genannt.
Eine Variable X wird in der Formel F als gebunden bezeichnet, wenn sie in einer Teilformel der Gestalt $\forall X G$ oder $\exists X G$ innerhalb von F auftritt. Andernfalls wird X als freie Variable in der Formel F bezeichnet.
Da die Formeln $\neg F \vee G$ und $(F \wedge G) \vee (\neg F \wedge \neg G)$ in der Logik besonders häufig auftreten, werden sie oft mit den Symbolen $\Longrightarrow$ und $\Longleftrightarrow$ abgekürzt.
Gemäß [Sch00b] (S. 14) wird die Formel $\neg F \vee G$ als $F \Longrightarrow G$ abgekürzt und die Formel $(F \wedge G) \vee (\neg F \wedge \neg G)$ als $F \Longleftrightarrow G$.
Diese Formeln werden auch als syntaktische Implikation und syntaktische Äquivalenz bezeichnet.

Semantik

Im folgenden Abschnitt wird die Semantik der Prädikatenlogik, also die Bedeutung der oben beschriebenen syntaktischen Elemente, festgelegt.
Die Semantik der Prädikatenlogik verleiht den syntaktischen Elementen ihre Bedeutung. Dies geschieht durch die Definition eines Universums in Form einer Menge, die Elemente enthält, welche in der Regel eine Domäne abbilden, sowie durch die Festlegung von Abbildungsvorschriften auf den Universum für jedes Funktionsymbol und die Zuweisung konkreter Werte aus dem Universum für jedes Variablensymbol. Auf diese Weise kann jedem Term ein eindeutiger Wert aus dem

Universum zugeordnet werden.
Prädikatensymbolen werden Wahrheitswerte zugeordnet, indem für jedes Prädikatensymbol eine Relation auf den Universum festgelegt wird. Auch den Operatorsymbolen wird eine Bedeutung zugewiesen, sodass jeder syntaktisch korrekten Formel ein eindeutiger Wahrheitswert zugeordnet werden kann.
Laut [Sch00b] (S. 52) wird für die Semantik eine Struktur benötigt, die sich aus dem Tupel $A = (U_A, I_A)$ zusammensetzt.
Diese ist wie folgt definiert:

Definition 2.3 (Struktur).
Eine Struktur $A = (U_A, I_A)$ besteht aus einer nicht leeren Grundmenge U_A von A, auch Universum genannt, und der Funktion I_A. I_A bildet ab von einer Teilmenge aus $\{P_i^k, f_i^k, X_i \mid i \in \mathbb{N} \text{ und } k \in \mathbb{N}\}$ auf eine Teilmenge von $\{P, f, X \mid P \subseteq U_A^k, f : U_A^k \to U_A, X \in U_A, k \in \mathbb{N}\}$.

Eine Struktur A wird als passend zu einer Formel F bezeichnet, falls I_A für alle in F vorkommenden Prädikatensymbole, Funktionssymbole und Variablen definiert ist mit passender Stelligkeit k.
Die Struktur weist folgenden syntaktischen Elementen aus der Prädikatenlogik eine Bedeutung zu:

- Jedem Funktionssymbol aus dem Definitionsbereich wird eine Funktion auf U_A zugeordnet mit gleicher Stelligkeit.
- Jedem Prädikatensymbol aus dem Definitionsbereich wird eine Relation auf U_A mit passender Stelligkeit zugeordnet.
- Jeder Variable aus dem Definitionsbereich wird ein Element aus U_A zugeordnet.

Mit der Struktur kann nun die Bedeutung von Termen definiert werden, indem sie induktiv ausgewertet und diesen Werten zugewiesen werden. Gemäß [Sch00b] (S. 53-54) werden Terme wie folgt induktiv ausgewertet:
Sei F eine Formel und A eine zu F passende Struktur.

- Falls X eine Variablensymbol in der Formel F ist, gilt:

$$A(X) = I_A(X)$$

- Falls f eine Funktionssymbol in der Formel F ist und die Form $f(t_1, \ldots, t_k)$ hat, gilt:

$$A(f) = I_A(f(A(t_1), \ldots, A(t_k)))$$

Mithilfe der Struktur und der Bedeutung von Termen können den Formeln Wahrheitswerte zugewiesen werden, indem die Formeln ebenfalls induktiv ausgewertet werden. Die induktive Auswertung von von Formeln erfolgt gemäß [Sch00b] (S. 54) wie folgt:
Sei F eine Formel und A eine zu F passende Struktur.

- Falls F ein Prädikat der Form $P(t_1, \ldots, t_k)$ ist, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls } (A(t_1), \ldots, A(t_k)) \in I_A(P) \\ 0 & \text{, sonst} \end{cases}$$

- Falls F die Form $\neg G$ hat, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls } A(G) = 0 \\ 0 & \text{, sonst} \end{cases}$$

- Falls F die Form $G \wedge H$ hat, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls } A(G) = 1 \text{ und } A(H) = 1 \\ 0 & \text{, sonst} \end{cases}$$

- Falls F die Form $G \vee H$ hat, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls } A(G) = 1 \text{ oder } A(H) = 1 \\ 0 & \text{, sonst} \end{cases}$$

- Falls F die Form $\exists X G$ hat, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls für alle } d \in U_A \text{ gilt } A[X/d](G) = 1 \\ 0 & \text{, sonst} \end{cases}$$

- Falls F die Form $\forall X G$ hat, gilt:

$$A(F) = \begin{cases} 1 & \text{, falls für ein } d \in U_A \text{ gilt } A[X/d](G) = 1 \\ 0 & \text{, sonst} \end{cases}$$

$A[X/d]$ ist wie folgt definiert: $A[X/d]$ ist eine neue Struktur, die identisch mit der Struktur A ist, außer für die Variablen, für die $X_i = X$ gilt. Diese werden neu zugeordnet mit $I_{A[X/d]}(X_i) = d$.
Unter Verwendung der festgelegten Bedeutungen kann nun festgestellt werden, ob zwei Formeln die gleiche Bedeutung haben. Dies wird auch als semantische Äquivalenz bezeichnet.
Laut [Sch00b] (S. 23) sind zwei Formeln F und G äquivalent, wenn für alle passenden Strukturen A gilt, dass $A(F) = A(G)$. Die entsprechende Schreibweise hierfür ist $F \equiv G$.
Des Weiteren kann bestimmt werden, ob es für eine Formel F eine passende Struktur A gibt, sodass diese wahr wird, ob es gar keine passende Struktur A gibt, sodass F wahr wird, oder ob die Formel F für jede beliebige passende Struktur A wahr ist. Dafür werden laut [Sch00b] (S. 55) folgende Begriffe und Notationen verwendet: Eine passende Struktur A, für die gilt $A(F) = 1$, wird auch als Modell der Formel F bezeichnet. Die entsprechende Notation dafür ist $A \models F$.

Wenn es mindestens eine passende Struktur A für eine Formel F gibt, sodass $A(F) = 1$ ist, wird die Formel als erfüllbar bezeichnet.
Falls es keine passende Struktur A gibt, für die $A(G) = 1$ gilt, wird eine Formel als unerfüllbar bezeichnet.
Wenn für jede passende Struktur A für eine Formel F gilt, dass $A(F) = 1$ ist, wird die Formel als gültig, allgemeingültig oder Tautologie bezeichnet.
Ein sehr wichtiger Bestandteil von Logiken ist die semantische Folgerung auch Deduktion genannt. Dabei wird eine Formel G als wahr aus einer Menge von Formeln F geschlossen.
Die Semantische Folgerung ist gemäß [Sch00b] (S. 19) wie folgt definiert:

Definition 2.4 (Folgerung).
Eine Formel G folgt aus den Formeln $F_1, \ldots, F_n$, wenn für jede Struktur A, die zu $F_1, \ldots, F_n$ und zu G passt, gilt: Wenn $F_1, \ldots, F_n$ unter A erfüllbar sind, dann ist auch G unter A erfüllbar.

Die Notation hierfür ist $F \models G$, wobei F für die Formelmenge $F = \{F_1, \ldots, F_n\}$ steht.
Sei $F = \{F_1, \ldots, F_n\}$ eine Menge von Formeln und G eine Formel. Dann ist nach [Sch00b] (S. 20) Folgendes äquivalent:
Falls gilt $n = 1$

- G ist eine Folgerung aus F_1
- $F_1 \implies G$ ist eine Tautologie
- $F_1 \wedge \neg G$ ist unerfüllbar

Falls gilt $n \geq 2$

- G ist eine Folgerung aus $F_1, \ldots, F_n$
- $(F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge F_n) \implies G$ ist eine Tautologie
- $(F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge F_n) \wedge \neg G$ ist unerfüllbar

Ein Literal L, wie in [Sch00b] (S. 38) definiert, ist entweder ein Prädikatensymbol P_i oder seine Negation $\neg P_i$, wobei $1 \leq i \leq n$ und $n \in \mathbb{N}$. Das bedeutet, dass L Teil der Menge $\{P_1, \ldots, P_n\} \cup \{\neg P_1, \ldots, \neg P_n\}$ ist.
Variablen können laut [Sch00b] (S. 61) in der Prädikatenlogik substituiert werden. Wenn F eine Formel ist, X eine Variable und t ein Term, dann bezeichnet $F[X/t]$ die Formel, die man erhält, wenn alle freien Vorkommen der Variablen X in F durch den Term t ersetzt werden.
Es ist auch möglich, eine Folge von Substitutionen zu definieren, die als $sub = [X_1/t_1] \ldots [X_n/t_n]$ beschrieben wird.
In der Literatur wird oft auch die alternative Notation σ für eine Folge von Substitutionen verwendet. Hierbei repräsentiert σ eine Abbildung, die durch $\sigma = [X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n]$ definiert ist, wobei jede Variable X_i auf den entsprechenden Term t_i abgebildet wird. Alle anderen Variablen werden auf sich selbst abgebildet (identische Abbildung). Die Anwendung der Substitution σ auf eine Formel F wird als $\sigma(F)$ dargestellt.
Unifikation bezeichnet gemäß [Sch00b] (S. 89) den Vorgang, Literale in eine gemeinsame Form zu bringen. Gegeben sei eine Menge von Literalen $L = \{L_1, \ldots, L_n\}$ und

eine beliebige Folge von Substitutionen *sub*. Wenn $L_1 sub = L_2 sub = \ldots = L_n sub$ gilt, ist L unifizierbar und *sub* ein Unifikator für L ist. Die entsprechende Schreibweise dafür ist auch $|Lsub| = 1$.
Eine Folge von Substitutionen *sub* einer Literalmenge L wird gemäß [Sch00b] (S. 89) als allgemeinster Unifikator bezeichnet, auch bekannt als Most General Unifier (MGU), wenn für jeden Unifikator sub' von L eine weitere Folge von Substitutionen *sab* existiert, sodass für jedes Literal $L_i \in L$ gilt: $L_i sub' = L_i subsab$.
In der Literatur wird der allgemeinste Unifikator häufig als $mgu(L_1, \ldots, L_n)$ bezeichnet.

2.1.2 TPTP (Thousands of Problems for Theorem Provers)

Laut [SS98] (S. 177) handelt es sich bei TPTP (Thousands of Problems for Theorem Provers) um eine Bibliothek von Problemen für automatische Theorembeweiser.
Die Probleme in der Bibliothek werden gemäß [SS98] (S. 181) in einem einheitlichen Format beschrieben, das in andere Formate umgewandelt werden kann.
Die vollständige Syntax des Formats ist in [tpt] spezifiziert. Im Folgenden wird eine einfache Erläuterung eines Teils der Syntax gegeben, der für diese Arbeit relevant ist, basierend auf der Spezifikation.
Die TPTP-Syntax ermöglicht die Beschreibung von Problemen in verschiedenen Sprachen. Daher ist es zunächst erforderlich, die Sprache anzugeben, in der das Problem beschrieben ist. In dieser Arbeit sind insbesondere die folgenden Sprachen von Bedeutung:
FOF (<name>, <formula_role>, <fof_formula><annotations>)
für Formeln in Prädikatenlogik erster Stufe und
CNF (<name>, <formula_role>, <cnf_formula><annotations>)
für Klauselnormalformen.
In folgender Tabelle 2.1 ist eine kurze Erläuterung der Bestandteile.

Elemente	**Bedeutung**
`<name>`	Name der Formel.
`<formula_role>`	Rolle der Formel: In dieser Arbeit sind insbesondere die Rollen *axiom* für ein Axiom und *conjecture* für eine zu beweisende Formel relevant.
`<fof_formula>`	Formel in Prädikatenlogik erster Stufe.
`<cnf_formula>`	Klauselnormalform.
`<annotations>`	Weitere beliebige Annotationen.

Tabelle 2.1: Elemente für die Definition einer Formel in TPTP-Syntax.

Im Folgenden soll die Syntax der Formeln weiter erläutert. Variablen werden in Formeln groß geschrieben, während Prädikate, Funktionen und Konstanten in der

Regel klein geschrieben werden. Des Weiteren sind die folgenden Operatoren definiert, die in der Tabelle 2.2 aufgeführt sind:

Operator in der TPTP-Syntax	**Operator in Prädikatenlogik**
&	$\wedge$
|	$\vee$
~	$\neg$
!	$\forall$
?	$\exists$
=>	$\Rightarrow$
<=>	$\Leftrightarrow$

Tabelle 2.2: Operatoren in der TPTP-Syntax und ihre äquivalenten prädikatenlogischen Operatoren.

In den Formeln sind die zwei Prädikate = für Gleichheit und ! = für Ungleichheit definiert. Zusätzlich sind die beiden speziellen Prädikate $true und $false spezifiziert, welche angeben, ob etwas wahr oder falsch ist.
Im Folgenden ist ein Beispiel für eine Formel in TPTP-Syntax und Prädikatenlogik:

Beispiel 2.5. $\text{fof}(\text{example\_formula}, \text{axiom}, (![X,Y] : (p(X)|q(Y)) \Rightarrow r(X,Y)))$

$$\forall X \forall Y \left((p(X) \vee q(Y)) \implies r(X,Y)\right)$$

2.1.3 Formale Theorien, Kalküle, Beweise

Im folgenden Abschnitt werden formale Theorien, Ableitungsregeln und Beweise behandelt. Laut [Dav89] (S. 2) setzt sich eine Theorie T aus den folgenden Komponenten zusammen:

- Eine abzählbare Menge von Symbolen S.
- Eine Menge Seq von Zeichenketten, die aus den Symbolen bestehen.
- Eine Teilmenge $WFF \subset Seq$, die wohldefinierte Formeln aus den Zeichenketten enthält.
- Eine Teilmenge $Axioms \subset WFF$ von Formeln, die auch Axiome genannt wird.
- Eine Menge R von Relationen über den Formeln. Die Relationen werden auch Ableitungsregeln genannt und die Menge R Kalkül.

Eine Formel $P \in WFF$ wird in einer Theorie T als ableitbar oder beweisbar aus einer Menge $M \subset WFF$ betrachtet laut [Dav89] (S. 4), wenn bestimmte Bedingungen erfüllt sind. Diese Bedingungen umfassen das Vorhandensein einer Sequenz $P_1, \ldots, P_n$, wobei $P_n = P$ und $1 < i < n$. Innerhalb dieser Sequenz kann ein P_i verschiedene Formen annehmen. Es kann entweder ein Axiom sein oder ein Element aus M (auch als Hypothese bezeichnet). Alternativ dazu kann P_i eine direkte

Konsequenz des Vorgängers von P_i sein. Diese Konsequenz entsteht durch Anwendung einer der Regeln aus R auf den Vorgänger von P_i.
Die beiden folgenden Eigenschaften beziehen sich auf Wahrheitswerte.
Nach [Dav89] (S. 6) ist eine Theorie vollständig, wenn alle allgemeingültigen Formeln auch beweisbar sind, und eine Theorie ist korrekt, wenn jeder beweisbare Satz auch allgemeingültig ist.

Beispiel 2.6. Angenommen, $WFF = \{A \wedge B, Q \vee \neg Q, C, D\}$ ist die Menge von wohlgeformten Formeln. $Axioms = \{A \wedge B\}$ ist die Menge der Axiome. Es gibt zwei Regeln: $A \wedge B \implies C$ und $C \implies D$. Somit kann D syntaktisch abgeleitet werden. Der Beweis basiert auf den Regeln $A \wedge B \implies C$ und $C \implies D$. Jedoch ist offensichtlich, dass das Kalkül nicht korrekt ist, da $A \wedge B \implies C$ keine Tautologie ist. Im Gegensatz dazu ist $Q \vee \neg Q$ eine Tautologie, aber nicht beweisbar. Daher ist die Theorie nicht vollständig.

2.1.4 Normalformen

Im folgenden Abschnitt werden Normalformen behandelt. Gemäß [Mel96] (S. 27) sind Normalformen syntaktisch standardisierte Formeln.
Viele moderne Beweiser transformieren als ersten Schritt Formeln in Normalformen, um die Erfüllbarkeit von Formeln einfacher zu zeigen.
Ein Nachteil von Normalformen besteht darin, dass sich die syntaktische Struktur der Formel in der Regel verändert, was sie für viele andere Anwendungen ungeeignet macht.
Für eine Normalform ist es wichtig, dass jede beliebige Formel mithilfe eines Algorithmus in diese umgewandelt werden kann.
Im Folgenden folgen die Definitionen einiger wichtiger Normalformen und Sätze zu diesen.
Nach [Sch00b] (S. 62) ist die Pränexnormalform (BPF) eine Formel, die die Form $Q_{1y_1} \ldots Q_{ny_n} F$ hat, wobei $Q_i \in \{\forall, \exists\}$ und $n \geq 0$ ist. Es ist wichtig anzumerken, dass für jede Formel F eine äquivalente (und bereinigte) Formel G in Pränexform existiert.
Gemäß [Sch00b] (S. 62) ist eine Formel bereinigt, wenn sie keine Variablen enthält, die sowohl gebunden als auch frei vorkommen, und wenn hinter allen Quantoren verschiedene Variablen stehen. Ebenso kann jede Formel in eine bereinigte Form umgeformt werden.
Sei $F = \forall_{y_1}, \ldots, \forall_{y_n} \exists_z G$ eine Pränexnormalform. Dann hat die Skolemnormalform gemäß [Sch00b] (S. 63-64) die Form $F' = \forall_{y_1} \ldots \forall_{y_n} F[z/f(y_1, \ldots, y_n)]$, wobei f ein neues, in F nicht vorkommendes n-stelliges Funktionsymbol ist. Es ist wichtig festzuhalten, dass „Jede Formel F in BPF ist genau dann erfüllbar, wenn die Skolemnormalform von F erfüllbar ist“ (vgl. [Sch00b], S. 64).
Zur Erzeugung der Skolem Normalform wird laut [Sch00b] (S. 64) die sogenannte Skolemisierung benötigt. Dabei werden die Existenzquantoren eliminiert, indem eine Skolem-Funktion f eingeführt wird, die von allen vorangehenden Allquantifizierten Variablen abhängt.
Eine Klauselnormalform hat nach [Sch00b] (S. 38), die Form $((L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge$

$\ldots \wedge (L_{k,1} \vee \ldots \vee L_{k,n_k}))$, wobei jedes $L_{i,j}$ ein Literal ist.
Eine alternative Darstellungsweise der Klauselnormalform besteht laut [Sch00b] (S. 38) darin, die Formel durch eine Menge von Klauseln auszudrücken. Diese sieht wie folgt aus:

$$F = \{\{L_{1,1}, \ldots, L_{1,n_1}\} \wedge \ldots \wedge \{L_{k,1}, \ldots, L_{k,n_k}\}\}$$

Die Elemente der Menge F werden auch als Klauseln bezeichnet.

2.1.5 Herbrand-Theorie

Im folgenden Abschnitt wird die Herbrand-Theorie beschrieben. Nach [Sch00b] (S. 73-74) ist diese Theorie von Bedeutung, da es in der allgemeinen Prädikatenlogik schwierig ist, die Erfüllbarkeit und Unerfüllbarkeit von Aussagen zu zeigen.
Dies liegt daran, dass bei der Auswahl des Universums sowie der Interpretation von Prädikaten und Funktionen ein hoher Grad an Freiheit herrscht.
Die Herbrand-Theorie beschränkt das Universum und die Interpretation, was die Analyse vereinfacht.

Definition 2.7 (Herbrand-Universum nach [Sch00b] (S. 77)).
Das Herbrand-Universum $H(F)$ für eine geschlossene Formel F in Skolemform ist wie folgt definiert:

- *Alle in F vorkommenden Konstanten sind Elemente von $H(F)$. Falls keine Konstante in F enthalten ist, wird eine beliebige Konstante a gewählt, die in $H(F)$ enthalten ist.*
- *Wenn $t_1, \ldots, t_n \in H(F)$ sind und f ein Funktionsymbol ist, das in F vorkommt, dann ist auch der Term $f(t_1, \ldots, t_n) \in H(F)$.*

Das bedeutet, dass das Herbrand-Universum alle variablenfreien Terme beinhaltet, die aus den Funktionssymbolen in F gebildet werden können.

Definition 2.8 (Herbrand-Struktur nach [Sch00b] (S. 78)).
Sei F eine Formel in Skolemform. Dann ist jede zu F passende Struktur $A = (U_A, I_A)$, für die folgendes gilt, eine Herbrand-Struktur für die Formel F:

- $U_A = H(F)$
- *Für jedes m-stellige Funktionssymbol f, das in F vorkommt, gilt $I_A(f(t_1, \ldots, t_n)) = f(t_1, \ldots, t_n)$, wobei $t_1, \ldots, t_n \in H(F)$.*

Die Struktur A wird auch Herbrand-Interpretation genannt. Es ist wichtig zu beachten, dass bei der Herbrand-Struktur jeder Term wieder auf sich selbst im Universum abgebildet wird.

Definition 2.9 (Herbrand-Expansion nach [Sch00b] (S. 81)).
Die Herbrand-Expansion einer Formel $F = \forall Y_1 \ldots \forall Y_n F^*$ in Skolemform definiert als $E(F) = \{F^*[Y_1/t_1] \ldots [Y_n/t_n] \mid t_1, \ldots, t_n \in H(F)\}$.

Das bedeutet, dass die Herbrand-Expansion jede Formel enthält, die entsteht, indem man die Variablen durch Terme aus dem Herbrand-Universum ersetzt. Die Expansion kann unendlich groß sein, falls das Herbrand-Universum unendlich ist. Wichtig ist, dass jede Formel $F \in E(F)$ keine Variablen mehr enthält, da diese durch Terme aus dem Herbrand-Universum ersetzt wurden.

2.1.6 Automatisiertes Beweisen

Im folgenden Abschnitt wird das automatisierte Beweisen behandelt.
Gemäß [BKKS96] (S. 1) ist logisches Schließen ein grundlegender Bestandteil menschlicher Problemlösungsstrategien. Daher zählt automatisiertes Beweisen (Automated Theorem Proving) gemäß [RN12] (S. 23) zu den Teilgebieten der Künstlichen Intelligenz.
Gemäß [BKKS96] (S. 1) werden automatische Theorembeweiser häufig dort eingesetzt, wo Probleme mittels mathematischer Logik spezifiziert werden können.
Anwendungsbereiche umfassen laut [BKKS96] (S. 1) beispielsweise das Beweisen mathematischer Theoreme, die Synthese von Computerprogrammen basierend auf einer Spezifikation sowie das Verhalten automatisierter Agenten unter bestimmten Rahmenbedingungen.
Das Ziel beim automatisierten Beweisen besteht darin, automatisch einen Beweis dafür zu finden, dass $\{F_1, \ldots, F_n\} \models G$ in einer Theorie T gilt. Die Axiome der Theorie T werden auch als Wissensbasis oder Knowledge Base bezeichnet, und die Formel G als Ziel. Viele Theorembeweiser nutzen die Tatsache aus, dass $\{F_1, \ldots, F_n\} \models G$ äquivalent dazu ist, dass $(F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge F_n) \wedge \neg G$ unerfüllbar ist.
Im Folgenden wird erläutert, wie automatisiert gezeigt werden kann, dass $(F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge F_n) \wedge \neg G$ unerfüllbar ist.
Laut [SS67] (S. 3) wird zur Vereinfachung des Suchraums in der Prädikatenlogik erster Stufe in der Regel auf Herbrand-Universen zurückgegriffen.
Auch zur Vereinfachung, wie von [Mel96] (S. 27) dargelegt, werden logische Formeln üblicherweise in eine Normalform umgewandelt, meistens in eine konjunktive Normalform (KNF), um sie algorithmisch einfacher zu verarbeiten.
Gemäß [RN12] (S. 305) wird durch wiederholtes Anwenden von Regeln eines Kalküls auf einer Wissensbasis versucht, einen Beweis zu finden.

Semientscheidbarkeit

Es stellt sich die Frage, ob überhaupt entschieden werden kann, ob $(F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge F_n) \wedge \neg G$ unerfüllbar ist.
Hierfür ist vor allem folgender Satz von Bedeutung:

Satz 2.10. *„Eine Formel F in Skolemform ist genau dann unerfüllbar, wenn es eine endliche Teilmenge von $E(F)$ gibt, die (im aussagenlogischen Sinne) unerfüllbar ist“ (vgl. [Sch00b], S. 82).*

Gemäß [Sch00b] (S. 82) kann ein Algorithmus erstellt werden, der bestimmt, ob eine Formel F unerfüllbar ist.

```
1 Eingabe: Formel F
2 Bringe die Formel F in Skolemform
3 Bilde eine beliebige Teilmenge {F_1, ..., F_n} ∈ E(F)
4 Überprüfe, ob F_1 ∧ ... ∧ F_n unerfüllbar ist
5 Falls F_1 ∧ ... ∧ F_n unerfüllbar ist, stoppe
6 Andernfalls gehe zu Zeile 3
```

Listing 2.1: Entscheidungsalgorithmus für die Prädikatenlogik erster Stufe, angelehnt an den Algorithmus aus [Sch00b], S. 82-83.

Der Algorithmus stoppt, wenn die Formel F unerfüllbar ist. Andernfalls läuft der Algorithmus endlos, da die Kardinalität von $E(F)$ unendlich sein kann. Somit ist das Problem semientscheidbar.

Superpositionskalkül

Moderne Beweiser greifen auf ein Kalkül namens Superpositionskalkül zurück.
Die Idee des Kalküls besteht darin, Prädikate mithilfe von Gleichheitsprädikaten zu repräsentieren. Gemäß [Mel96] (S. 71-72) besitzen Gleichheitsrelationen die fundamentalen Eigenschaften der Symmetrie, Reflexivität und Transitivität. Darüber hinaus ermöglicht die Gleichheitsrelation die Substitution von Gleichem durch Gleiches.
Der Umgang mit Gleichheitsprädikaten entspricht grundsätzlich dem Umgang mit anderen Prädikaten in der Prädikatenlogik.
Im Folgenden werden einige Notationen und Erläuterungen aus [Sch02] (S. 2-3) zur besseren Verständlichkeit des Superpositionskalküls beschrieben.
Für beliebige Terme s und t ist die Notation für das Gleichheitsprädikat $s \simeq t$ und für das Ungleichheitsprädikat $s \not\simeq t$. Seien u und s Terme, wobei p ein Subterm von u ist, dann bezeichnet $u[p \leftarrow s]$ den Term, der entsteht, wenn der Subterm p in u durch den Term s ersetzt wird. Der Subterm p in u wird mit $u|_p$ bezeichnet.
Eine Ordnung auf Termen und Klauseln kann dabei helfen, die Anwendung der Regeln des Superpositionskalküls auf bestimmte Situationen zu beschränken. Aus diesem Grund wird eine Ordnung $>$ zur Vereinfachung eingeführt.
Diese hat laut [Sch02] (S. 2) die Eigenschaft, dass sie total auf allen variablenfreien Termen ist und stabil in Bezug auf die Termstruktur und Substitution ist.
Das bedeutet gemäß [Sch00a] (S. 16), dass bei der Substitution eines Terms durch einen anderen die Ordnung erhalten bleibt. Anders ausgedrückt, wenn für die Terme t_1, t_2 die Beziehung $t_1 > t_2$ gilt, dann gilt auch $\sigma(t_1) > \sigma(t_2)$ für eine Substitution σ.
Des Weiteren wird durch die Ordnung die Monotonie beibehalten. Gemäß [Sch00a] (S. 16) bedeutet dies, dass für Terme t_1, t_2, s_1, s_2 und p gilt: Wenn $s_1 > s_2$ ist, dann muss auch $t_1[p \leftarrow s_1] > t_2[p \leftarrow s_2]$ gelten.
Eine Multimenge einer Menge M ist gemäß [Sch00a] (S. 8) eine Funktion $A : M \to \mathbb{N}$, für die gilt, dass die Menge $\{x \mid A(x) > 0\}$ endlich ist. In der Regel werden Multimengen definiert, indem Elemente in einer Mengennotation nummeriert werden oder indem einfach mehrfach Vorkommen eines Elements in der Mengennotation vorgenommen werden. Alle möglichen Multimengen über M werden durch

Mult(M) beschrieben. Gemäß [Sch00a] (S. 8-9) können auf Multimengen ähnliche Operationen wie auf Mengen definiert werden. Für Multimengen A und B über M lassen sich beispielsweise folgende Definitionen aufstellen:

- $x \in A$ falls $A(x) > 0$
- $A \subseteq B$ falls $A(x) \leq B(x)$ für alle $x \in M$
- $(A \cup B)(x) = A(x) + B(x)$ für alle $x\ inM$
- $(A \setminus B)(x) = \max(A(x) - B(x), 0)$ für alle $x \in M$

Beispiel 2.11. Einige Beispiele:
Seien $M = \{a, b, c\}$, $A = \{a : 1, b : 2, c : 0\}$ und $B = \{a : 2, b : 2, c : 1\}$. Dann gilt:

- $a : 1 \in A$
- $A \subseteq B$, da $A(a) < B(a), A(b) < B(b), A(c) < B(c)$
- $(A \cup B) = \{a : 3, b : 4, c : 1\}$
- $(A \setminus B) = \{a : 0, b : 0, c : 0\} = \emptyset$

Gemäß [Sch00a] (S. 9) kann die oben definierte Ordnung $>$ auf endliche Multimengen erweitert werden, indem eine neue Ordnung $>>$ auf Multimengen definiert wird. Sei M eine Menge, $A, B \in \text{Mult}(M)$ und sei $>$ eine wie oben definierte Ordnung auf der Menge M. Dann gilt $A >> B$, wenn Mengen $X, Y \in \text{Mult}(M)$ existieren mit $X \neq \emptyset$, $X \subseteq A$, $B = (A \setminus X) \cup Y$ und für alle $y \in Y$ ein $x \in X$ existiert mit $x > y$.
Ein Beispiel für die Ordnung aus [Sch00a] (S. 9) ist:

Beispiel 2.12. Sei $M = \{a, b, c, d\}$ eine Menge mit $a > b > c > d$. Seien $A = \{a, b, c\}$, $B = \{b, b, b, b, b, c, c, d, d, d\}$ Multimengen. Dann gilt $A >> B$ da $B = (A \backslash a) \cup \{b, b, b, b, c, d, d, d\}$ und $a > b$, $a > c$, $a > d$.

Die Ordnung auf Multimengen kann gemäß [Sch02] (S. 2) auch auf Literale übertragen werden, da Literale mittels Multimengen dargestellt werden können. Zum Beispiel kann $s \simeq t$ über die Multimenge $\{\{s\}, \{t\}\}$ und $s \not\simeq t$ über die Multimenge $\{s, t\}$ dargestellt werden. Durch die Verwendung von Multimengen ist es möglich, die Literale zu vergleichen. Gleiches gilt für Klauseln, da diese als Multimengen von Literalen repräsentiert werden können und somit ebenfalls geordnet werden können. Die Klauseln werden im Folgenden mit C dargestellt. C^- bezeichnet die negativen Literale in der Klausel C, während C^+ die positiven Literale in der Klausel C beschreibt.
Die Selektionsfunktion *sel* dient gemäß [Sch02] (S. 3) dazu, eine Menge von Literalen aus einer Klausel zu selektieren. Eine Selektionsfunktion hat folgende Eigenschaften, die für jede Klausel C gelten müssen:

- $sel(C) \subseteq C$
- Wenn $sel(C) \cap C^- = \emptyset$, dann ist $sel(C) = \emptyset$. Das bedeutet, dass $sel(C)$ mindestens ein negatives Literal enthält oder leer ist.

Eine Substitution wird mit dem Buchstaben σ abgekürzt, und der allgemeinste Unifikator von Termen s und t mit $mgu(s, t)$.

Zuletzt müssen noch Regeln definiert werden, wann welche Regel des Kalküls angewendet werden darf.
Regel 1:
Für $\sigma(l)$ muss Folgendes gelten:

- $sel(C) = \emptyset$ und $\sigma(l)$ ist maximal in C,
- $sel(C) \neq \emptyset$ und $\sigma(l)$ ist maximal in $\sigma(sel(C) \cap C^-)$,
- $sel(C) \neq \emptyset$ und $\sigma(l)$ ist maximal in $\sigma(sel(C) \cap C^+)$.

Regel 2:
Für $\sigma(l)$ muss Folgendes gelten:

- $l \in C^+$, $sel(C) = \emptyset$ und $\sigma(l)$ ist maximal in C.

Seien R und S Teilklauseln, und seien s, t, u, v und p Terme. Dann lauten die Regeln des Superpositionskalküls nach [Sch02] (S. 3) wie folgt:

Equality Resolution

$$\frac{s \not\simeq t \vee R}{\sigma(R)} \tag{2.1}$$

Wobei gilt: $\sigma = \mathrm{mgu}(s, t)$ und $\sigma(s \not\simeq t)$ erfüllt die Regel 1.

Beispiel 2.13. Ein Beispiel für die Regel unter der Annahme, dass die Bedingungen für die Anwendung gelten, wäre: Sei $\{n(A) \not\simeq n(b), g(c) \simeq q(A)\}$ eine Klausel. Dann kann daraus die Klausel $\{g(c) \simeq q(b)\}$ geschlossen werden unter Verwendung des $mgu = [A \leftarrow b]$.

Superoposition into negative literals

$$\frac{s \simeq t \vee S \qquad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \tag{2.2}$$

Wobei gilt: $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ erfüllt die Regel 2, $\sigma(u \not\simeq v)$ erfüllt die Regel 1 und $u|_p \notin V$.
Beispiel 2.14. Ein Beispiel für die Regel unter der Annahme, dass die Bedingungen für die Anwendung gelten, wäre: Sei $\{n(Z) \simeq g(h), p(c) \simeq q(Z)\}$ eine Klausel und $\{r(n(l)) \not\simeq r(d), n(X) \simeq g(c)\}$ eine weitere Klausel. Dann kann daraus die Klausel $\{r(g(h)) \not\simeq r(d), p(c) \simeq q(l), n(X) \simeq g(c)\}$ geschlossen werden unter Verwendung des $mgu = [Z \leftarrow l]$.

Superoposition into positive literals

$$\frac{s \simeq t \vee S \qquad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)} \tag{2.3}$$

Wobei gilt: $\sigma = mgu(u|_p, s)$, $\sigma(s) \not\preceq \sigma(t)$, $\sigma(u) \not\preceq \sigma(v)$, $\sigma(s \simeq t)$ erfüllt die Regel 2, $\sigma(u \not\simeq v)$ erfüllt die Regel 1 und $u|_p \notin V$.
Beispiel 2.15. Ein Beispiel für die Regel unter der Annahme, dass die Bedingungen für die Anwendung gelten, wäre: Sei $\{n(Z) \simeq g(h), p(c) \simeq q(Z)\}$ eine Klausel und $\{r(n(l)) \simeq r(d), n(X) \simeq g(c)\}$ eine weitere Klausel. Dann kann daraus die Klausel $\{r(g(h)) \simeq r(d), p(c) \simeq q(l), n(X) \simeq g(c)\}$ geschlossen werden unter Verwendung des $mgu = [Z \leftarrow l]$.

Equality factoring

$$\frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)} \tag{2.4}$$

Wobei gilt: $\sigma = mgu(s, u)$, $\sigma(s) \not\preceq \sigma(t)$ und $\sigma(s \simeq t)$ erfüllt die Regel 2.
Beispiel 2.16. Ein Beispiel für die Regel unter der Annahme, dass die Bedingungen für die Anwendung gelten, wäre: Sei $\{n(Z) \simeq p(g), n(c) \simeq q(g), m(d) \not\simeq p(Z)\}$ eine Klausel. Dann kann daraus die Klausel $\{p(g) \not\simeq q(g), n(c) \simeq q(g), m(d) \not\simeq p(c)\}$ geschlossen werden unter Verwendung des $mgu = [Z \leftarrow c]$.

Der Kalkül kann jedoch durch weitere Regeln erweitert werden, zum Beispiel um doppelte Literale in einer Klausel zu eliminieren, triviale Ungleichungen der Form $s \not\simeq s$ zu entfernen, Tautologien in Klauseln zu beseitigen, Subsumptionslösungen durchzuführen und Terme umzuschreiben, um sie durch kleinere Terme zu ersetzen.

Given-Clause Algorithmus

Der Given-Clause-Algorithmus dient dazu, die Erfüllbarkeit oder Unerfüllbarkeit prädikatenlogischer Formeln mithilfe des Superpositionskalküls zu zeigen. Dadurch kann der Algorithmus für automatisiertes Beweisen verwendet werden. Im Folgenden wird der Given-Clause Algorithmus genauer erläutert.
Der Given-Clause Algorithmus gehört laut [SM16] (S. 331) zu den sogenannten Saturationsalgorithmen. Das bedeutet, dass in jedem Schritt mittels Regeln neue Klauseln hergeleitet und Klauseln vereinfacht werden. Dies geschieht solange, bis entweder keine neuen Klauseln hergeleitet werden können, die nicht redundant sind und somit die Erfüllbarkeit gezeigt ist, oder die leere Klausel abgeleitet ist und dadurch die Unerfüllbarkeit gezeigt ist. Da die Prädikatenlogik erster Stufe unentscheidbar ist, kann es auch vorkommen, dass der Algorithmus unendlich läuft.
Saturationsalgorithmen organisieren laut [SM16] (S. 332-333) den Suchraum, also alle möglichen Folgerungen aus der Klauselmenge.
Es gibt zwei verschiedene Strategien, dies zu tun. Die erste besteht darin, eine

Klausel C_i aus einer Klauselmenge M_i zu selektieren und dann alle möglichen Folgerungen $F_1 \dots F_n$ aus M_i zu bilden, wobei C_i als Prämisse verwendet wird. Die Folgerungen werden dann zu M_i hinzugefügt, und man erhält die neue Klauselmenge $M_{i+1} = M_i \cup \{F_1, \dots, F_n\}$. Diese Methode wird auch als Level-Saturation bezeichnet.
Ein Problem bei der Level-Saturation besteht darin, dass man die Suche nicht wirklich durch eine Heuristik beeinflussen kann.
Eine andere Möglichkeit ist ein Single-Step-Algorithmus. Dieser wählt eine Klausel C_i aus einer Klauselmenge M_i aus und bildet eine Folgerung F_1 aus M_i, wobei C_i als Prämisse verwendet wird. Diese Folgerung wird dann der Klauselmenge M_i hinzugefügt, und es entsteht eine neue Klauselmenge $M_{i+1} = M_i \cup \{F_1\}$. Der Vorteil dieser Methode besteht darin, dass die Suche sehr fein granuliert mit einer Heuristik beeinflusst werden kann. Der Nachteil ist jedoch, dass der Protokollaufwand sehr hoch ist. Moderne Beweiser nutzen laut [SM16] (S. 333) varianten des Given-Clause Algorithmuses. Bei diesem wird die Klauselmenge in zwei Mengen aufgeteilt. Die Menge der verarbeiteten Klauseln und die Menge der unverarbeiteten Klauseln.
In jedem Schritt wird hierbei eine Klausel g aus der Menge der unverarbeiteten Klauseln selektiert. Dann werden alle Folgerungen gebildet die aus der Menge der verarbeiteten Klauseln gebildet werden können und die g als Prämisse haben. Diese werden dann zu der menge der unverarbeiteten Klauseln hinzugefügt.
Es gibt zwei Varianten des Given-Clause Algorithmus: der Otter-Loop und der Discount-Loop. Beim Otter-Loop werden sowohl Klauseln aus der Menge der unverarbeiteten Klauseln als auch aus der Menge der verarbeiteten Klauseln zur Vereinfachung verwendet. Beim Discount-Loop werden nur Klauseln aus der Menge der verarbeiteten Klauseln zur Vereinfachung genutzt.

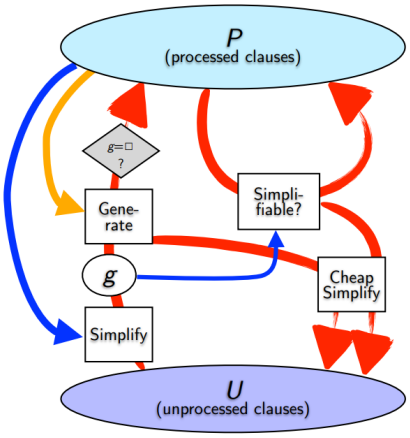


Abbildung 2.1: Grafische Darstellung des Given-Clause Algroithmus des E Prover aus [SUP], Folie 46.

```
1   while U ≠ ∅
2      g = extract_best(U)
3      g = simplify(g, P)
4      if g is the empty clause then
5           SUCCESS, Proof found
6      if g isn't subsumed by any clause in P (or otherwise redundant w.r.t P)
7           P = P\{c ∈ P | c subsumed by (or otherwise redundant w.r.t) g}
8           T = {c ∈ P | c can be simplified with g}
9           P = (P\T) ∪ {g}
10          T = T ∪ generate(g, P)
11          foreach c ∈ T
12               c = cheap_simplify(c, P)
13               if c is not trivial
14                    U = U ∪ {c}
15  SUCCESS, Original U is satisfiable
```

Listing 2.2: Given-Clause Algorithmus des E Prover aus [SM16], S. 332.

Im Algorithmus werden laut [SM16] (S. 333) zwei Mengen verwendet: U und P. Die Menge U enthält alle noch nicht verarbeiteten Klauseln, während die Menge P alle bereits verarbeiteten Klauseln enthält.

Der Algorithmus wird ausgeführt, bis entweder alle Klauseln aus U verarbeitet wurden (Code Zeile 1) oder bis ein Beweis gefunden wurde beziehungsweise die leere Klausel hergeleitet wurde (Code Zeilen 4-5).

In Zeile 2 des Codes wird nach [Sch02] (S. 7) zunächst die beste Klausel ausgewählt mithilfe der Funktion extract_best(U). Eine detaillierte Erläuterung zur Auswahlstrategie und dazu, wie die beste Klausel bestimmt wird, folgt später.
Dies ist von besonderer Bedeutung, da eine verbesserte Auswahl der Klauseln, beispielsweise durch den Einsatz einer Heuristik, dazu beiträgt, die Anzahl der Schritte zur Ableitung der leeren Klausel zu reduzieren. Dadurch können Beweise in weniger Schritten gefunden werden.

In Zeile 3 des Codes wird laut [Sch02] (S. 7) die ausgewählte Klausel g mithilfe der bereits verarbeiteten Klauseln aus P in der Funktion simplify(c, P) vereinfacht, sofern dies möglich ist. Hierfür werden Regeln aus dem Superpositionskalkül verwendet.

In Zeile 6 des Codes wird laut Ausage von [Sch02] (S. 7) überprüft, ob eine Klausel in P die ausgewählte Klausel g subsumiert oder ob sie überflüssig ist. Dabei kommen Regeln aus dem Superpositionskalkül zum Einsatz. Sollte eine Subsumption oder Überflüssigkeit festgestellt werden, wird zum Code in Zeile 2 zurückgesprungen, um eine neue Klausel auszuwählen.

In Zeile 7 des Codes werden dann alle Klauseln aus P entfernt, die durch Regeln aus dem Superpositionskalkül von der ausgewählten Klausel g subsumiert oder als überflüssig erkannt werden.

In Zeile 8 des Codes wird eine neue Hilfsmenge eingeführt, welche Klauseln c

enthält, die durch Anwendung von g vereinfacht werden können.

In Zeile 9 des Codes werden die Klauseln aus P, die durch c vereinfacht werden können, entfernt. Danach wird die Klausel g zur Menge P der verarbeiteten Klauseln hinzugefügt.

In Zeile 10 des Codes werden nach [Sch02] (S. 7) alle Klauseln, die mittels g aus der Menge P neu erzeugt werden können, zur Menge T hinzugefügt. Dies erfolgt mithilfe der Funktion generate(g, P). Insbesondere sind dafür die Regeln der Equality Resolution, Superposition into negative literals und Superposition into positive literals sowie die Regel Equality Factoring aus dem Superpositionskalkül verantwortlich.

In Zeile 11 des Codes wird jede Klausel c aus der Menge T durchlaufen.

In Zeile 12 des Codes wird gemäß [Sch02] (S. 7) die Klausel c gegebenenfalls mithilfe der Klauseln aus der Menge P in der Funktion cheap_simplify(c, P) vereinfacht. Hierbei kommen Regeln aus dem Superpositionskalkül zum Einsatz.

In Zeile 13 des Codes wird [Sch02] (S. 7) zufolge überprüft, ob die Klausel c in Bezug auf die Menge P mittels Regeln aus dem Superpositionskalkül redundant ist. Falls nicht, wird die Klausel c zu der Menge der unverarbeiteten Klauseln U hinzugefügt.

Gemäß [SM16] (S. 333) ist ein Ordnung der Terme und somit die Auswahl des Literals aus der Klausel im Given-Clause Algorithmus vorgegeben. Die Auswahl einer Klausel aus der Menge der unverarbeiteten Klauseln ist jedoch frei wählbar. Im folgenden Abschnitt wird die Auswahl der Klausel genauer erläutert.
Die meisten Implementierungen ordnen jeder Klausel ein heuristisches Gewicht zu. Die Klauseln werden dann in absteigender Gewichtsfolge bearbeitet, wobei die Klausel mit dem geringsten Gewicht immer zuerst behandelt wird.
Im Folgenden werden einige Heuristiken entsprechend [SM16] (S. 333-334) erläutert:

- First-in/First-out (FIFO): Hier wird immer die älteste unverarbeitete Klausel bevorzugt.
- Symbol Counting: Die Anzahl der Variablen und Funktionssymbole in den Klauseln wird gezählt. Dadurch werden kleinere Klauseln bevorzugt, die weniger Symbole enthalten.
- Ordering-Aware: Diese Variante von Symbol Counting bevorzugt Klauseln mit weniger Literalen oder Termen mit maximalem Gewicht.

Bekannte Heuristiken berücksichtigen jedoch nicht die semantische Bedeutung der Symbolnamen bei der Auswahl einer Klausel. In dieser Arbeit wird dies durch die Verwendung eines neuronalen Netzes umgesetzt, das die Eignung eines Symbols

für ein zu beweisendes Ziel bewertet. Basierend auf dieser Bewertung werden dann die Klauseln ausgewählt.

2.2 Neuronale Netze

In diesem Abschnitt wird eine kurze theoretische Einführung in neuronale Netze gegeben. Es werden ihre Anwendungsbereiche (siehe Abschnitt 2.2.1) sowie die grundlegenden Bestandteile von neuronalen Netzen (siehe Abschnitt 2.2.2) kurz erläutert. Das Training neuronaler Netze mithilfe des Backpropagation-Algorithmus wird in Abschnitt 2.2.3 behandelt. Im Abschnitt 2.2.5 werden Large Language Models vorgestellt, die im Verlauf dieser Arbeit verwendet werden.

2.2.1 Anwendungsbereich von Neuronalen Netzen

Neuronale Netze sind nach [Cal03] (S. 15) Verarbeitungseinheiten, die aus vielen miteinander verbundenen Neuronen bestehen. Ähnlich wie klassische Algorithmen werden laut [Cal03] (S. 16) Neuronale Netze zur Lösung von Problemen eingesetzt. Der wesentliche Unterschied besteht jedoch darin, dass bei neuronalen Netzen Lösungen nicht programmiert, sondern erlernt werden können. Besonders in Bereichen, in denen es schwierig oder sogar unmöglich ist, eine Lösung zu programmieren, sind neuronale Netze von großer Bedeutung. Es gibt auch Bereiche, in denen die erlernten Lösungen von neuronalen Netzen schlichtweg besser sind als klassische Lösungen.

2.2.2 Bestandteile von Neuronalen Netzen

Im Folgenden werden die einzelnen Bestandteile von neuronalen Netzen kurz erläutert.

Neuronen, Signale und Gewichte

Ein neuronales Netz besteht gemäß [Cal03] (S. 17) aus Neuronen, auch Knoten genannt, die miteinander gerichtet verbunden sind. Jeder Knoten kann eingehende und ausgehende Verbindungen haben. Diese Verbindungen haben sogenannte Gewichte.
Das Gewicht eines Neurons wird laut [Cal03] (S. 18) durch drei Parameter bestimmt: die herführende Einheit, die hinführende Einheit und den Gewichtswert selbst.
Die Funktion eines Neurons, nach [Cal03] (S. 17), liegt darin, eingehende Signale zu verarbeiten und daraus ausgehende Signale zu berechnen. Wie in [Cal03] (S. 19) erläutert wird, bestimmt der absolute Gewichtswert die Stärke der Verbindung. Ein positiver eingehender Gewichtswert hat dabei eine anregende Wirkung, während ein negativer eingehender Gewichtswert die Verarbeitung des Eingangssignals hemmt.
Es existieren nach [Cal03] (S. 18) zwei spezielle Arten von Neuronen: Eingabeeinheiten, die Signale von außerhalb des neuronalen Netzes empfangen, und Ausgabeinheiten, die Signale vom neuronalen Netz nach außen geben.

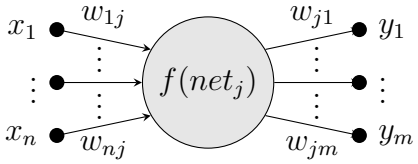
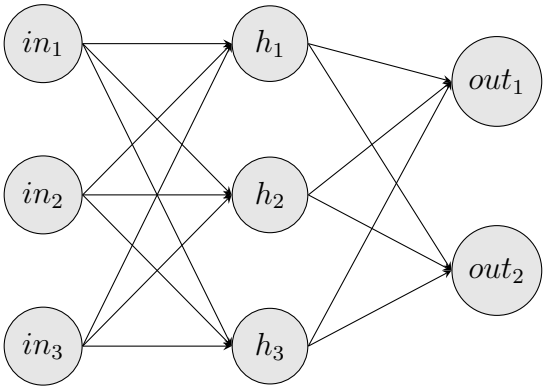


Abbildung 2.2: Darstellung eines Neurons basierend auf Abbildung 1.1 in [Cal03] (S. 17).

In der Abbildung 2.2 sind $x_1, \ldots, x_n$ die eingehenden Signale und $w_{1j}, \ldots, w_{nj}$ die Gewichte an den eingehenden Verbindungen zum Neuron j. Das berechnete eingehende Signal, das aus allen eingehenden Signalen berechnet wird, ist net_j, und f ist die sogenannte Aktivierungsfunktion, die das Ausgangssignal berechnet. Die Gewichte der ausgehenden Verbindungen sind $w_{j1}, \ldots, w_{jm}$, und $y_1, \ldots, y_m$ sind die neuen Eingangssignale für die nächsten Neuronen am Ende der ausgehenden Verbindungen.

Konnektivitätsmuster

Laut [Cal03] (S. 18) wird die Art und Weise, wie Neuronen miteinander verbunden sind und welche Gewichte die Verbindungen haben, durch das sogenannte Konnektivitätsmuster festgelegt. Dabei gibt es verschiedene Arten von Strukturen, wie Neuronen miteinander verbunden sind. Nach [Cal03] (S. 19) ist eine häufige Anordnung in Schichten. Bei dieser Art der Anordnung wird die erste Schicht als Eingabeschicht und die letzte Schicht als Ausgabeschicht bezeichnet. Die Schichten zwischen der Eingabe- und Ausgabeschicht werden auch als verborgene Schichten bezeichnet. Es gibt jedoch auch zahlreiche weitere Möglichkeiten der Verbindung.



Anordnung der Neuronen in Schichten
basierend auf Abbildung 1.4 in (vgl. [Cal03], S. 20).

Die Gewichte des Konnektivitätsmusters werden nach [Cal03] (S. 18) in der Regel durch das sogenannte Training festgelegt. Dabei lernt das Netz, das Lösen eines Problems durch Anpassen der Gewichte. Das Training des Netzes wird im Abschnitt 2.2.3 näher erläutert. Nach [Cal03] (S. 19) werden die Gewichte und Verbindungen typischerweise mit einer sogenannten Adjazenzmatrix modelliert,

welche auch als Gewichtsmatrix bezeichnet wird. Im Folgenden ist eine generelle Adjazenzmatrix für n Neuronen abgebildet:

$$\text{Adj} = \begin{bmatrix} w_{00} & w_{01} & w_{02} & \dots & w_{0n} \\ w_{10} & a_{11} & w_{12} & \dots & w_{1n} \\ w_{20} & w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{n1} & a_{n2} & \dots & w_{nn} \end{bmatrix}$$

Dabei steht der Eintrag w_{jk} in der Matrix in Zeile j und Spalte k für das Gewicht an der Verbindung von Neuron j zum Neuron k.

Signalausbreitung im Netz

Es muss laut [Cal03] (S. 20) festgelegt werden, wann Einheiten aktualisiert werden, d.h., wann Eingangssignale kombiniert, und ein Ausgangssignal berechnet wird und wann ein Signal an eine andere Einheit gesendet werden darf. Auf diese Weise wird die Ausbreitung des Signals im Netz gesteuert, und die Neuronen werden in der richtigen Reihenfolge aktualisiert. Es kann zum Beispiel eine Aktualisierung in Gruppen erfolgen, sodass eine Gruppe von Neuronen nach der anderen im Netz aktualisiert wird.

Verarbeitung von Eingangssignalen

Für jedes Neuron muss nach [Cal03] (S. 20) festgelegt werden, wie die Eingangssignale zu einem Gesamteingangssignal verarbeitet werden. Hierfür wird eine Funktion benötigt, die k Eingangswerte entgegennimmt und daraus einen einzelnen Eingangswert net_j für das Neuron j berechnet . Im Folgenden werden zwei Methoden zur Berechnung vorgestellt.
Die erste Methode besteht gemäß [Cal03] (S. 21) darin, die einzelnen Signale zu summieren und mit den Gewichten zu multiplizieren. Die Formel hierfür lautet:

$$net_j = \sum_{i=1}^{k-1} (w_{ij} * x_i) \tag{2.5}$$

Ein Vorteil dieser Methode besteht darin, dass sie auch mittels Matrixmultiplikation wie folgt berechnet werden kann:

$$net_j = \begin{bmatrix} w_{0j} \dots w_{kj} \end{bmatrix} * \begin{bmatrix} x_0 \\ \vdots \\ x_k \end{bmatrix} \tag{2.6}$$

Eine alternative Methode besteht nach [Cal03] (S. 21) darin, für jedes Neuron die Differenz zwischen dem Eingangssignal und dem Gewicht zu berechnen und diese Werte dann zu quadrieren und aufzusummieren. Die entsprechende Formel lautet:

$$net_j = \sum_{i=0}^{k-1} (w_{ij} - x_i)^2 \tag{2.7}$$

In den obigen Formeln steht x_i für das eingehende Signal des Neurons i und w_{ij} für das Gewicht der Verbindung zwischen Neuron i und Neuron j.

Berechnung des Ausgangssignals

Für jedes Neuron muss ebenfalls laut [Cal03] (S. 21) festgelegt werden, wie das Eingangssignal, das sich aus den vielen Eingangssignalen zusammensetzt (wie in Abschnitt 2.2.2 beschrieben), zu einem Ausgangssignal verarbeitet werden soll. Dies geschieht mithilfe einer Aktivierungsfunktion. Der von dieser Funktion berechnete Ausgangswert wird als Aktivierung bezeichnet. Im Folgenden werden einige wichtige Aktivierungsfunktionen beschrieben.

Identitätsfunktion

Eine der bedeutendsten Aktivierungsfunktionen ist nach [Cal03] (S. 21-22) die Identitätsfunktion, bei der die Eingabe gleich der Ausgabe entspricht. Diese Funktion findet insbesondere Anwendung bei Einheiten, bei denen der Wert der Eingabe gleich dem Wert der Aktivierung sein soll. Dies ist üblicherweise bei Eingabe- und Ausgabeeinheiten der Fall.

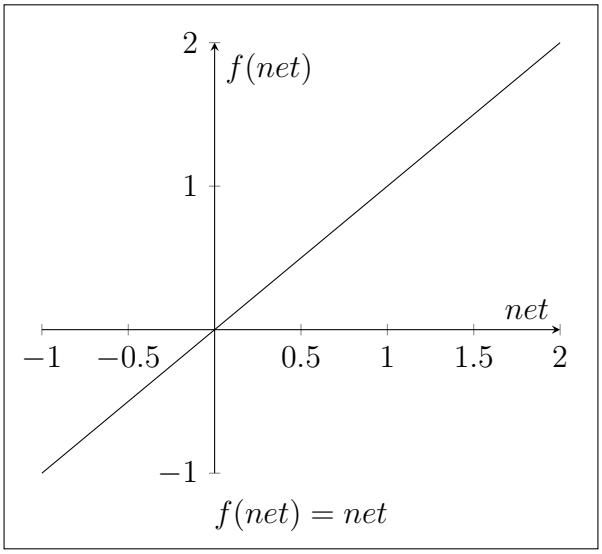


Abbildung 2.3: Graph der Identitätsfunktion basierend auf Abbildung 1.6 aus [Cal03] (S. 22).

Schwellenwertfunktion

Eine weitere wichtige Aktivierungsfunktion ist [Cal03] (S.22) zufolge die Schwellenfunktion. Die Schwellenfunktion aktiviert ein Neuron basierend auf dem Verhältnis

von Netzeingang zum Schwellenwert.
Eine wichtige Schwellenfunktion ist die Binäre Schwellenfunktion. Diese beschränkt die Aktivierung des Neurons auf eins oder null, je nach dem Verhältnis des Netzeingangs zum Schwellenwert.

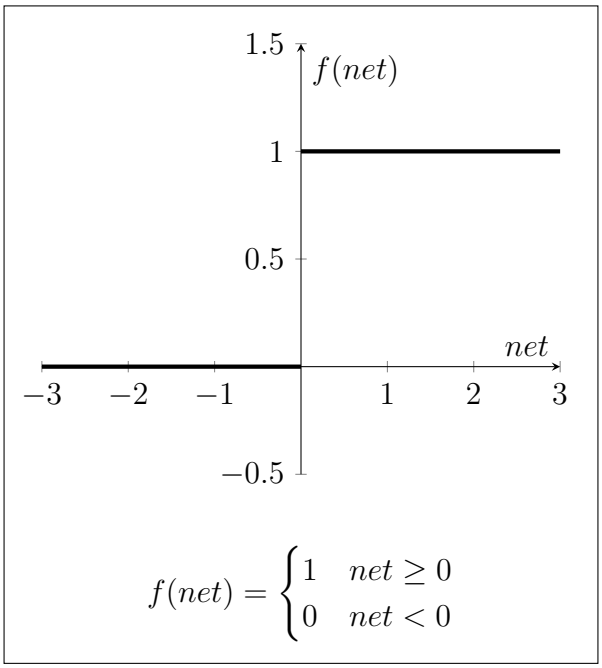


Abbildung 2.4: Graph der binären Schwellenwertfunktion basierend auf Abbildung 1.8 aus [Cal03] (S. 22).

Gemäß [Cal03] (S.22) werden Schwellenfunktionen oftmals implementiert, indem ein Verschiebeterm zu dem verarbeiteten Signal hinzugefügt wird, welches in 2.2.2 beschrieben wird. Dieser Verschiebeterm wird durch Addition eines negativen Schwellenwerts zum Eingangssignal erreicht.
Für die Methode des Aufaddierens der k Eingangssignale des Neurons j, die mit den jeweiligen Gewichten multipliziert werden, sieht dies beispielsweise wie folgt aus:

$$net_j = w_0 + \sum_{i=0}^{k-1} (w_{ij} * x_i) \tag{2.8}$$

Der Verschiebungsterm (im Englischen Bias) kann laut [Cal03] (S. 23) auch als Gewicht betrachtet werden, das von einer Einheit ausgeht. Diese Einheit hat konstant den Wert eins als Eingangssignal, und das entsprechende Gewicht dieses Eingangssignals ist dann der Verschiebungsterm. Bei dieser Implementierung bleibt die Berechnung des Eingangssignals unverändert.

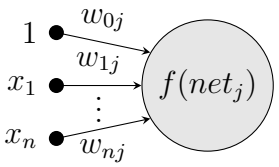


Abbildung 2.5: Abbildung eines Neurons mit Verschiebungsterm w_{0j} in Anlehnung an Abbildung 1.9 aus [Cal03] (S. 23).

Sigmoidfunktion

Eine besonders wichtige Aktivierungsfunktion ist nach [Cal03] (S.23) die Sigmoidfunktion. Sie nimmt ein beliebiges Eingangssignal und berechnet ein Ausgangssignal zwischen null und eins.

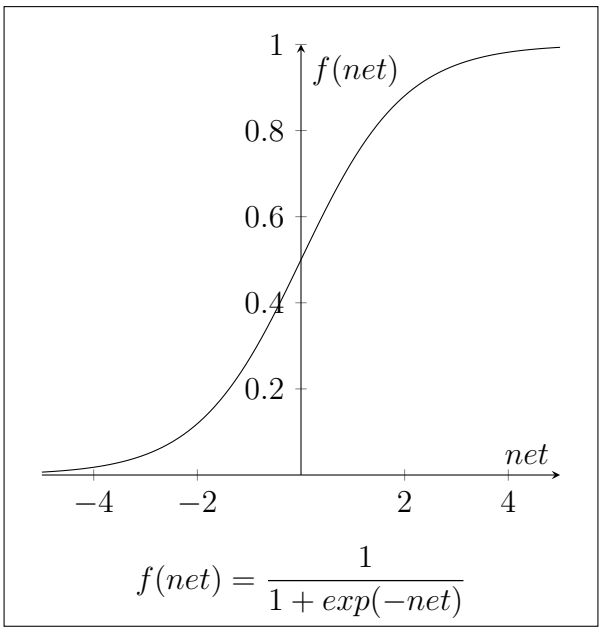


Abbildung 2.6: Graph einer Sigmoidfunktion basierend auf Abbildung 1.10 in [Cal03] (S. 23).

2.2.3 Training neuronaler Netze

Im Folgenden wird das Training von neuronalen Netzen erläutert und wie ein neuronales Netz durch dieses Training lernt.
Mit „Lernen“ wird gemäß [RN12] (S. 809) gemeint, dass die Leistung für zukünftige Aufgaben verbessert wird. Es gibt verschiedene Formen des Lernens, die unterschieden werden können. Eine häufige Lernform ist laut [RN12] (S. 811) das

überwachte Lernen. Hierfür benötigt man eine Menge von N Eingabe-Ausgabe-Beispiel-Paaren $(x_1, y_1), \ldots, (x_N, y_N)$. Die Ausgabebeispiele y_i sind durch eine unbekannte Funktion $y = f(x)$ generiert worden. Ziel ist es nun, mittels einer Funktion $h : X \to Y$ die Funktion f anzunähern.
Die Funktion h wird gemäß [RN12] (S. 812) als Hypothese bezeichnet. Eine Hypothese ist gut, wenn sie die Daten, die von f generiert wurden, gut vorhersagen kann.
In der Regel sollen jedoch auch Vorhersagen für neue Beispiele getroffen werden können, die nicht in den Trainingsdaten enthalten sind. Dafür werden die Trainingsdaten gemäß [RN12] (S. 812) in zwei Mengen aufgeteilt: eine zur Erstellung der Hypothese h und eine weitere, um die Genauigkeit der Hypothese zu messen.
Ist Y eine unendliche Menge, so wird gemäß [RN12] (S. 812) das Problem des Findens der Hypothese h als Regressionsproblem bezeichnet. Ist Y eine endliche Menge, handelt es sich laut [RN12] (S. 812) um ein Klassifikationsproblem.
In dieser Arbeit wurde ein Sprachmodell trainiert, das die Eignung eines Symbols (Funktions-, Prädikaten- oder Konstantensymbol) für ein gegebenes Beweisziel bewertet. Hierbei wird ein Bereich von reellen Zahlen von 1 bis -1 für die Bewertung verwendet. Somit handelt es sich um ein Regressionsproblem.
Um die passenden Gewichte eines neuronalen Netzwerks zu finden, damit es die geeignete Hypothese h liefert, muss zunächst eine Bewertungsfunktion definiert werden. Diese beurteilt, ob die Ausgabe des neuronalen Netzwerks den gewünschten Anforderungen entspricht.
Dies geschieht gemäß [Cal03] (S.26) mit einer Fehlerfunktion, die die Diskrepanz zwischen der Ausgabe und der Soll-Ausgabe angibt.
Eine beliebte Methode ist der sogenannte quadratische Fehler.
Der quadratische Fehler für ein Trainingsdatum T ist gemäß [Cal03] (S. 49) wie folgt definiert:

$$E_T = \frac{1}{2} \sum_{j=1}^{N} (y_j - o_j)^2 \tag{2.9}$$

Hierbei steht N für die Anzahl der Ausgabeeinheiten, y_j repräsentiert die angestrebte Ausgabe des Ausgabeneurons j bei der Netzwerkeingabe x, und o_j beschreibt die tatsächliche Ausgabe des Ausgabeneurons j unter der Netzwerkeingabe x.
Der Gesamtfehler E des Netzes kann gemäß [Cal03] (S. 40) berechnet werden, indem alle E_T aufsummiert werden, also $E = \sum E_T$. Im Folgenden wird es darum gehen, den Gesamtfehler zu minimieren, indem die Gewichte des Netzes entsprechend angepasst werden.
Eine Methode zur Anpassung der Gewichte und zum Minimieren des Gesamtfehlers ist der Backpropagation-Algorithmus. Dabei wird der Gradient der Fehlerfunktion bezüglich der Gewichte berechnet und mit dem Gradientenabstiegsverfahren die Fehlerfunktion bezüglich der Gewichte minimiert.
Um den Fehler in Bezug auf die Gewichte mittels des Gradientenverfahrens zu minimieren, muss die partielle Ableitung bezüglich der Gewichtsvariable gebildet werden.

Mithilfe der Kettenregel kann die Fehlerfunktion gemäß [Cal03] (S. 50-51) wie folgt abgeleitet werden:

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta o_j} \cdot \frac{\delta o_j}{\delta net_j} \cdot \frac{\delta net_j}{\delta w_{ij}} \tag{2.10}$$

Dabei sind $E_T = \frac{1}{2}\sum_{j=1}^{N}(y_j - o_j)^2$, $o_j = f(net_j)$ und $net_j = \sum_{i=0}^{k-1}(w_{ij} * x_i)$ wie bereits oben definiert. Die Ableitungen ergeben sich wie folgt:

$$\frac{\delta E}{\delta o_j} = -(y_j - o_j) \tag{2.11}$$

$$\frac{\delta o_j}{\delta net_j} = f'(net_j) \tag{2.12}$$

$$\frac{\delta net_j}{\delta w_{ij}} = x_i \tag{2.13}$$

Für die partielle Ableitung der Fehlerfunktion wird die Ableitung der Aktivierungsfunktion des Neurons benötigt.
Häufig wird als Ableitung für die Aktivierungsfunktion eine Sigmoid-Funktion eingesetzt. Die allgemeine Ableitung einer Sigmoid-Funktion f lautet gemäß [Cal03] (S. 52) wie folgt:

$$f'(net_j) = f(net_j) \cdot [1 - f(net_j)] \tag{2.14}$$

Basierend auf den obigen Ableitungen lässt sich ein Term ϵ bestimmen, der beschreibt, wie eine Änderung eines eingehenden Signals eines Ausgabeneurons j den Output beeinflusst und welchen Beitrag der Output zur Änderung des Gesamtfehlers leistet.
Dieser ist gemäß [Cal03] (S. 51) wie folgt definiert:

$$\epsilon_j = -\frac{\delta E}{\delta o_j} \cdot \frac{\delta o_j}{\delta net_j} = (y_j - o_j) \cdot f'(net_j) \tag{2.15}$$

Für verdeckte Einheiten kann der Term nicht unmittelbar bestimmt werden. Stattdessen wird der Fehler im Netzwerk rückwärts propagiert, da der Fehler von allen Fehlern der vorherigen Einheiten abhängt. Dieser Prozess wird gemäß [Cal03] (S. 52) wie folgt beschrieben:

$$\epsilon_j = f'(net_j) \cdot \sum_k \epsilon_k \cdot w_{kj} \tag{2.16}$$

Die Variable k bezeichnet die vorherige Schicht, die für die Rückpropagierung des Fehlers verantwortlich ist. Die Fehler in der vorherigen Schicht werden jeweils mit dem entsprechenden Gewicht multipliziert und summiert. Der Fehler einer verdeckten Einheit hängt also von den vorherigen Fehlern und den Gewichten ab. Somit ist die Ableitung des Fehlers bezüglich eines Gewichts gemäß [Cal03] (S. 51-52) wie folgt definiert:

$$\frac{\delta E}{\delta w_{ij}} = \epsilon_j \cdot x_i \tag{2.17}$$

Die Gewichte werden mithilfe des Gradientenabstiegsverfahrens so angepasst, dass der Gesamtfehler reduziert wird. Dabei muss beschrieben werden, wie die Gewichte von Trainingsdatum zu Trainingsdatum angepasst werden, um den Gesamtfehler zu verringern. Die entsprechende Formel lautet gemäß [Cal03] (S. 54) wie folgt:

$$\Delta w_{ij}(n+1) = \mu \cdot (\epsilon_j \cdot x_i) + \alpha \cdot \Delta w_{ij}(n) \tag{2.18}$$

In dieser Formel steht n für das Trainingsdatum, μ für die sogenannte Lernrate und α für einen sogenannten Trägheitsterm, der einen gewissen Anteil der vorherigen Gewichtsänderung hinzufügt. Auf diese Weise werden zu große Schwankungen der Gewichtsänderung vermieden. In der Regel findet das Training in sogenannten Epochen statt. Eine Epoche ist hierbei laut [Cal03] (S. 50) das einmalige Präsentieren aller Trainingsdaten. Ein neuronales Netz konvergiert nach [Cal03] (S. 50), wenn der Betrag des mittleren quadratischen Fehlers zwischen zwei Epochen innerhalb eines bestimmten Toleranzbereichs liegt. Es existieren jedoch auch weitere alternative Konvergenzkriterien. In der Regel wird das Training oder die Anpassung der Gewichte beendet, sobald das Netz konvergiert.
Vor dem Training des neuronalen Netzes müssen gemäß [Cal03] (S. 27) folgende Schritte durchgeführt werden:

1. Auswahl eines geeigneten Netzwerktyps.
2. Festlegung einer passenden Netztopologie.
3. Spezifikation der zu erlernenden Parameter, darunter Gewichte und Schwellenwerte.

Der vollständige Ablauf des Backpropagation-Algorithmus, basierend auf [Cal03] (S. 53), gestaltet sich wie folgt:
Vor Beginn des Trainings werden die Gewichte des neuronalen Netzes zufällig initialisiert.

1. Ein Trainingsdatum, bestehend aus Eingabe x und entsprechender Soll-Ausgabe y, wird eingelesen.
2. Für jedes Neuron, das in der Eingabeschicht ist, wird die Eingabe und die Ausgabe gemäß berechnet:
$$\begin{aligned} \text{net}_j &= x_j \\ o_j &= f(\text{net}_j) \end{aligned}$$
3. Für jedes Neuron, das nicht in der Eingabeschicht ist, wird die Eingabe und die Ausgabe gemäß berechnet:
$$\begin{aligned} \text{net}_j &= \textstyle\sum_{i=1}^{n} x_i \cdot w_{ij} \\ o_j &= f(\text{net}_j) \end{aligned}$$
4. Für jedes Neuron in der Ausgabeschicht wird der Fehler berechnet:
$$\epsilon_j = -(y_j - o_j) \cdot f'(\text{net}_j)$$
5. Für jedes Neuron in den verborgenen Schichten wird der Fehler berechnet:
$$\epsilon_j = f'(\text{net}_j) \cdot \sum_k \epsilon_k \cdot w_{kj}$$

6. Alle Gewichte werden aktualisiert gemäß:

$$\Delta w_{ij}(n+1) = \mu \cdot (\epsilon_j \cdot x_i) + \alpha \cdot \Delta w_{ij}(n)$$

7. Einlesen des nächsten Trainingsdatums und erneuter Durchlauf ab Schritt 1.

Zur Vereinfachung wurde ein Abbruchkriterium für die Beendigung des Trainings im obigen Ablauf des Algorithmus weggelassen. Im Folgenden ist eine erstellte Implementierung in Pseudocode, welche eine Abbruchbedingung beinhaltet.

```
1  def backpropagation(training_data, criteria):
2      # Initialisierung der Abbruchbedingung
3      criteria_met = False
4
5      while not criteria_met:
6          for trainingData in training_data:
7
8              # Einlesen der Trainingsdaten
9              inputs, targets = trainingData
10
11             # Zurücksetzen des Kriteriums für jeden Durchlauf
12             criteria_met = True
13
14             # Vorwärtspfad
15
16             # Eingabeneuronen initialisieren
17             for each neuron in first_layer:
18                 # Berechnung der Netzwerkeingabe
19                 net_j = inputs[j]
20                 # Berechnung der Netzwerkausgabe
21                 o_j = f(net_j)
22
23             for each neuron in hidden_layers:
24                 # Berechnung der Netzwerkeingabe
25                 net_j = sum_{i=1}^{n} x_i * w_ij
26                 # Berechnung der Netzwerkausgabe
27                 o_j = f(net_j)
28
29             # Überprüfen, ob das Kriterium erfüllt ist
30             if not criteria_is_met(criteria):
31                 criteria_met = False
32
33             # Rückwärtspfad
34
35             for each neuron in output_layer:
36                 epsilon_j = -(targets[j] - o_j) * f'(net_j)
37
38             for each neuron in hidden_layers:
39                 epsilon_j = f'(net_j) * sum_k epsilon_k * w_kj
40
41             # Aktualisiere Gewichte
42             for each neuron:
43                 Delta w_ij(n+1) = mu * (epsilon_j * x_i) + alpha * Delta w_ij(n)
```

Listing 2.3: Backpropagation Algorithmus basierend auf [Cal03], S. 53.

Der Algorithmus aktualisiert nach jeder Präsentation eines Trainingsdatums die Gewichte. Es existieren jedoch laut[RN12] (S. 838) auch andere Herangehensweisen, wie beispielsweise das Arbeiten mit Trainingsdaten in Chargen (Batches), bei dem der Gradient gemittelt und die Gewichte nach jeder Batch-Arbeitung aktualisiert werden, oder der stochastische Gradientenabstieg. Bei letzterem wird nach

jeder Epoche der gemittelte Gradient zur Aktualisierung der Gewichte verwendet. Die Verarbeitung in Batches hat allerdings den Nachteil, dass sie möglicherweise sehr langsam konvergiert, während der stochastische Gradientenabstieg mitunter gar nicht konvergiert.

2.2.4 Evaluation des Trainings von neuronalen Netzen

Die Evaluation der Performance von neuronalen Netzen ist ein komplexes Thema, das durch zahlreiche Metriken beeinflusst wird. Eine zentrale Frage ist, wann das Training beendet werden sollte, um optimale Vorhersageergebnisse zu erzielen. Ein niedriger Trainingsfehler garantiert nicht notwendigerweise eine hohe Vorhersagequalität des Netzes.
Zuvor wurde bereits das Konvergenzkriterium eingeführt, welches jedoch nicht immer geeignet ist, da es beispielsweise zu Overfitting führen kann. Gemäß [BG21] (S. 6391) bedeutet Underfitting, dass ein neuronales Netz nicht in der Lage ist, die Beziehungen in den Daten zu erlernen, was zu unzuverlässigen Vorhersagen führt. Overfitting, das genaue Gegenteil, tritt auf, wenn das Netz die Trainingsdaten zu gut lernt und dadurch eine Generalisierung auf neue Daten verhindert wird. In beiden Fällen sind die Vorhersagen ungenau.
Eine Methode, dies zu verhindern, besteht darin, verschiedene Modelle zu vergleichen, wie auf Seite 6400 in [BG21] erwähnt wird. Modelle mit geringem Fehler auf den Testdaten, niedrigem Bias (unverzerrten Vorhersagen) und geringer Varianz führen zu genaueren Vorhersagen.
In dieser Arbeit wurde das Modell in Epochen trainiert, wobei jede trainierte Epoche als eigenständiges Modell betrachtet werden kann. Für jedes Modell wurden der durchschnittliche absolute Fehler (Mean Absolute Error, MAE), die Standardabweichung der absoluten Fehler und der mittlere Fehler (Mean Bias Error, MBE) berechnet.
Gemäß [RW23] (S. 2) ist der MAE definiert als:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |P_i - O_i|$$

wobei P_i der vom Netz geschätzte Wert und O_i der tatsächliche Wert ist.
Der MBE wird gemäß [RW23] (S. 3) definiert als:

$$MBE = \overline{P} - \overline{O}$$

wobei $\overline{P}$ das arithmetische Mittel der geschätzten Werte und $\overline{O}$ das arithmetische Mittel der tatsächlichen Werte ist.
Der MAE quantifiziert den durchschnittlichen Fehler zwischen den geschätzten und den tatsächlichen Werten. Die Standardabweichung der absoluten Fehler misst die Streuung um den Mittelwert. Eine große Streuung deutet auf hohe Fehlerabweichungen hin, während eine geringe Streuung auf eine geringe Abweichung der Fehler hinweist. Der MBE gibt an, wie stark das Netz die tatsächlichen Werte im Durchschnitt über- oder unterschätzt.

In dieser Arbeit wurden die Werte mittels der Bootstrapping-Methode ermittelt. Bootstrapping adressiert das Problem von Ausreißern in Datensätzen, die statistische Berechnungen wie das arithmetische Mittel verzerren können. Dieser Ansatz eignet sich gemäß [Rei09] (S. 533) für die Analyse von Regressionsmodellen auf Basis von Residuen.
Das Verfahren funktioniert folgendermaßen: Zunächst werden alle Residuen (die Differenzen zwischen Schätzwerten und tatsächlichen Werten) für die Testdaten berechnet. Dann werden n Residuen zufällig mit Zurücklegen gezogen, was eine Stichprobe bildet. Für jede Stichprobe wird der gewünschte statistische Wert (z.B. MAE, MBE oder die Standardabweichung der absoluten Fehler) berechnet. Auf diese Weise entsteht eine neue Verteilung dieser statistischen Werte.
Der Mittelwert dieser neuen Verteilung kann dann verwendet werden, um eine Näherung des tatsächlichen Wertes zu ermitteln. Dadurch wird eine robustere Schätzung ermöglicht, die weniger anfällig für die Verzerrungen durch Ausreißer ist.
Das neuronale Netz wurde in dieser Arbeit über mehrere Epochen hinweg trainiert. Die Auswahl des optimalen neuronalen Netzes erfolgte anhand der Epoche, die bezüglich des Mittelwerts des absoluten Fehlers, der Standardabweichung der absoluten Fehler und des mittleren Fehlers die besten Ergebnisse erzielte.

2.2.5 Unterschiedliche Arten von neuronalen Netzen

Es existieren diverse Arten von neuronalen Netzen, die für unterschiedliche Probleme eingesetzt werden.
In dieser Arbeit wird ein Large Language Model (LLM) verwendet. Aus diesem Grund werden im Folgenden Large Language Models näher erläutert.

Große Sprachmodelle

Große Sprachmodelle (Large Language Models, abgekürzt LLMs), laut [Ama23] (S. 82-83), sind neuronale Netze, die in der Lage sind, Sprache zu verarbeiten.
Sie verfügen über eine enorme Anzahl von Parametern, die trainiert werden können. Diese Modelle werden mit umfangreichen Datensätzen trainiert und sind daher äußerst vielseitig einsetzbar.
Die Vielzahl der Parameter ermöglicht es dem Netz, laut [Ama23] (S. 82), komplexere Aufgaben zu lösen. Allerdings erfordern große Sprachmodelle im Allgemeinen erhebliche Rechenleistung.
Durch die sogenannte Feinabstimmung (im Englischen fine-tuning) können große Sprachmodelle angepasst werden, wie in [Ama23] (S. 107) erläutert. Dabei kann das Wissen, das das Modell über eine bestimmte Aufgabe hat, auf eine andere Aufgabe durch Training übertragen werden, um dem Modell das Lösen neuer Aufgaben beizubringen.
Im Rahmen dieser Arbeit wurde ein LLM feinabgestimmt (im Englischen fine-tuned), um zu bewerten, wie gut ein Symbol zu einer Beweisaufgabe passt. Architekturen für LLMs sind laut [Ama23] (S. 90) unter anderem Transformer-Modelle

und rekurrente neuronale Netze, welche im Verlauf der Arbeit noch näher beschrieben werden.

Repräsentation von Texten

Es stellt sich die Frage, wie Text in Zahlen kodiert werden kann, sodass er von neuronalen Netzen verarbeitet werden kann.
Gemäß [Ama23] (S. 23) wird hierfür zunächst der Text in sogenannte Tokens aufgeteilt, also in kleinere Einheiten.
Eine Methode laut [Ama23] (S. 20) ist die White-Space-Tokenisierung, bei der die Wörter an den Leerzeichen getrennt werden.
Alle einzigartigen Tokens bilden gemäß [Ama23] (S. 23) das sogenannte Vokabular. Diesen Tokens muss nun ein eindeutiger numerischer Wert zugewiesen werden, welcher dann für das Encoding verwendet werden kann. Eine Möglichkeit hierfür ist laut [PC20] (S. 4-5) die sogenannte One-Hot-Repräsentation. Ähnlich wie bei ASCII werden dabei Wörter mittels Binärzahlen codiert. Dabei enthält das zu kodierende Wort an einer bestimmten Stelle eine Eins, während alle anderen Stellen mit Nullen belegt werden.

Beispiel 2.17. Als Beispiel können ein Vokabular basierend auf den Wörtern „Eis“, „Spaghetti“ und „Brot“ mit den Binärzahlen (1,0,0), (0,1,0) und (0,0,1) kodiert werden.

Eine flexiblere Möglichkeit nach [PC20] (S. 5-6) bietet das Vektorraummodell (VRM). Dabei werden Wörter oder andere Objekte in einen n-dimensionalen Raum kodiert. Diese Repräsentation hat unter anderem den Vorteil, dass Distanzen zwischen Objekten existieren, um beispielsweise Ähnlichkeiten darzustellen. Zudem können sehr viele Objekte in einem Raum mit sehr wenigen Dimensionen dargestellt werden.

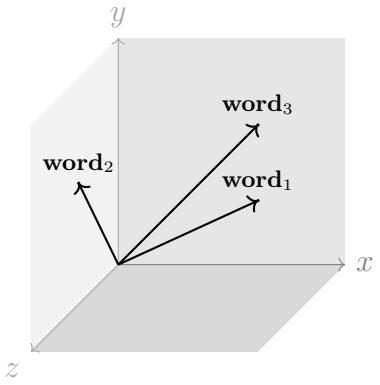


Abbildung 2.7: Dreidimensionales Vektorraummodell basierend auf der Abbildung 1.2 aus [PC20] (S. 6).

Verarbeitung von Sequenzen

Es stellt sich die Frage, wie Sequenzen unterschiedlicher Längen effektiv in einem neuronalen Netzwerk verarbeitet werden können.
In den folgenden Zeichnungen bezeichnet der Parameter „t“ einen Zustand zu einem bestimmten Zeitpunkt t.
Eine Möglichkeit sind laut [PC20] (S. 12-13) rekurrente neuronale Netzwerke (RNNs). Der Aufbau von RNNs ist recht einfach: Es wird lediglich eine Rückkopplungsschleife in das Netzwerk integriert, um den Ausgabewert des vorherigen Wortes auf gewisse Weise zu berücksichtigen.

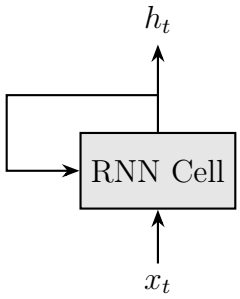


Abbildung 2.8: RNN-Zelle basierend auf der Abbildung 2.1 aus [PC20], S. 13.

Wie in Abbildung 2.8 sehen kann, wird der Output h erneut als Input in die Zelle gegeben zusammen mit dem Input x. In der folgenden Abbildung 2.9 ist zu erkennen, wie sich der Zustand des RNN über die Zeit hinweg verändert.

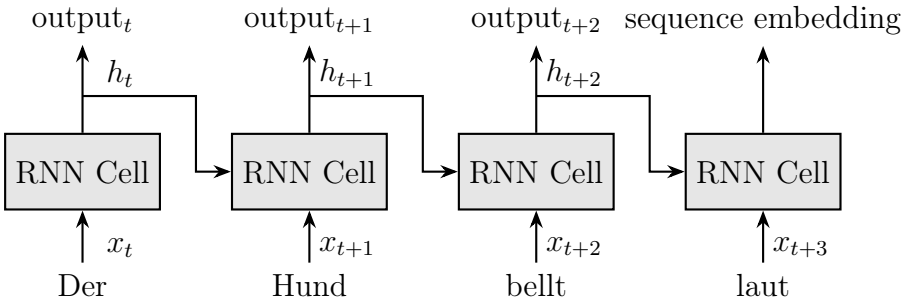


Abbildung 2.9: RNN-Zelle, die sich im Laufe der Zeit verändert, gemäß Abbildung 2.1 in [PC20] (S. 13).

In der Abbildung 2.9 wird der Satz „Der Hund bellt“ schrittweise in die RNN-Zelle eingegeben. Dabei wird der vorherige Output zusammen mit dem nächsten Wort als Input verwendet. Auf diese Weise merkt sich die Zelle gewissermaßen den vorherigen Zustand, wodurch beliebig lange Wortsequenzen kodiert werden können. Ein Problem, das jedoch laut [PC20] (S. 14-15) auftritt, ist der „Vanishing Gradient“. Bei sehr kleinen Gewichtswerten wird der Gradient durch die Wiederholung

des Feedback-Loops immer kleiner. Auf diese Weise ist die Anzahl der Wiederholungen begrenzt, da das neuronale Netz ab einer bestimmten Länge nicht mehr trainiert werden kann.
Eine verbesserte Variante gemäß [PC20] (S. 15) ist das sogenannte Long Short-Term Memory (LSTM), das versucht, dieses Problem zu umgehen beziehungsweise die Anzahl der möglichen Wiederholungen zu vergrößern. Im Folgenden ist eine Long Short-Term Memory-Zelle zu sehen.

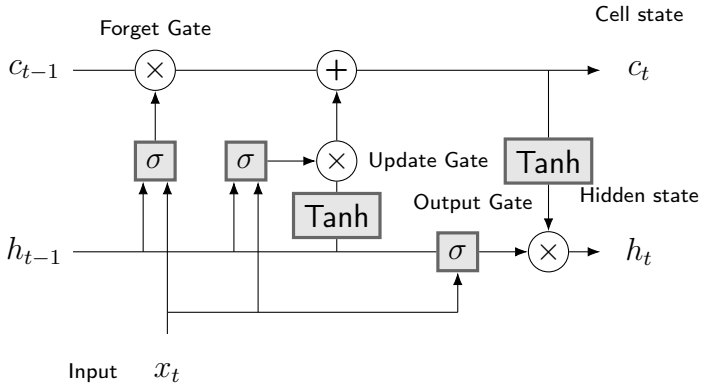


Abbildung 2.10: LSTM-Zelle gemäß Abbildung 2.4 in [PC20] (S. 16).

Eine Long Short-Term Memory (LSTM)-Zelle, wie in Abbildung 2.10 dargestellt, besteht gemäß [PC20] (S. 16-17) aus drei verschiedenen Gates, die den Zellspeicher c beeinflussen.
Das Forget-Gate bestimmt mithilfe der Sigmoid-Funktion, wie viel des Speichers vergessen werden soll. Das Update-Gate bestimmt über die Sigmoid-Funktion, wie viel im Speicher aktualisiert werden soll. Das Output-Gate steuert mithilfe der Sigmoid-Funktion, wie viel des Speichers für den Output und somit für den nächsten Teil der Sequenz verwendet werden soll.

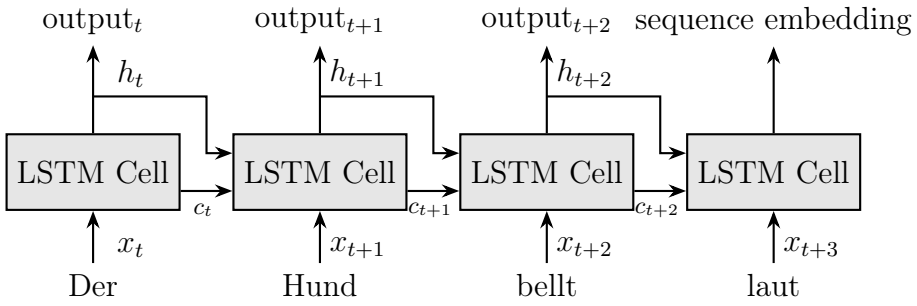


Abbildung 2.11: LSTM-Zelle, die sich im Laufe der Zeit verändert, gemäß Abbildung 2.3 in [PC20] (S. 15).

In der Abbildung 2.11 wird die zeitliche Veränderung einer LSTM-Zelle dargestellt. Das Grundprinzip ähnelt dem einer RNN-Zelle, jedoch wird hier zusätzlich der Zellzustand abgebildet.
Alle bisher beschriebenen Ansätze erzeugen gemäß [PC20] (S. 17) einen Output fester Länge. Jedoch ist es oft erforderlich, für Eingaben unterschiedlicher Länge einen entsprechend variablen Output zu generieren. Modelle, die diese Anpassungsfähigkeit bieten, werden als sogenannte Sequence-to-Sequence (Seq2Seq) Modelle bezeichnet. Dies ist insbesondere bei Übersetzungen relevant, da übersetzte Sätze in der Zielsprache unterschiedliche Längen im Vergleich zur Ausgangssprache haben können. Das deutsche Wort „Handy“ wird im Englischen beispielsweise als zwei Wörter „mobile phone“ übersetzt. Eine beliebte Form von Seq2Seq-Modellen sind gemäß [PC20] (S. 18-19) sogenannte Encoder-Decoder-Modelle.
Im Folgenden wird die Funktionsweise eines Encoder-Decoder-Modells auf Basis von [PC20] (S. 18-19) erläutert.
Der Encoder verarbeitet eine Input-Sequenz $x = (x_1, \ldots, x_n)$ zu einer Output-Sequenz $r = (r_1, \ldots, r_n)$.
Der Decoder verarbeitet anschließend die Output-Sequenz $r = (r_1, \ldots, r_n)$ zu einer Output-Sequenz beliebiger Länge $y = (y_1, \ldots, y_m)$. Die Länge der Input-Sequenz x mit n Elementen und die der Output-Sequenz y mit m Elementen können unterschiedlich sein. Dadurch können beliebig lange Sequenzen als Output generiert werden, unabhängig von der Länge des Inputs.
Der Decoder wird mit dem letzten Output des Encoders und einem Start-Token initialisiert.
Unter Verwendung des Einbettungsvektors und des Start-Tokens werden solange Outputs generiert, bis der Decoder ein sogenanntes Stopp-Token erzeugt.
Für den Encoder und den Decoder können gemäß [PC20] (S. 18) beispielsweise zwei RNN-Zellen verwendet werden.
Im Folgenden ist eine Abbildung eines einfachen Encoder-Decoder-Modells zu sehen.

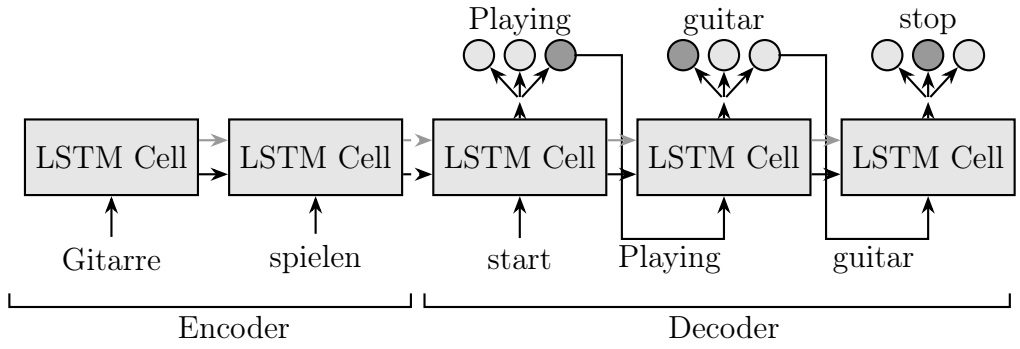


Abbildung 2.12: Encoder-Decoder-Modell gemäß Abbildung 2.6 [PC20] (S. 19).

In Abbildung 2.12 ist ein Beispiel eines Encoder-Decoder-Modells zur Übersetzung natürlicher Sprache dargestellt. Der Encoder kodiert die Wörter „Gitarre“

und „spielen“ in einen Einbettungsvektor.
Der Decoder beginnt mit dem vordefinierten Start-Token und dem Einbettungsvektor, um die Wörter zu decodieren und die Übersetzung der Wörter im Beispiel „Playing“ und „guitar“ anhand von Wahrscheinlichkeiten zu ermitteln, bis das Stopp-Token erreicht wird.
Auf diese Weise kann ein neuronales Netzwerk nahezu beliebige Sätze übersetzen. Ein Problem gemäß [PC20] (S. 19) besteht darin, dass alle Wörter vom Encoder in einen einzigen Einbettungsvektor codiert werden. Eine Möglichkeit, dies zu umgehen, ist der sogenannte Attention-Mechanismus, der in der folgenden Abbildung dargestellt ist.

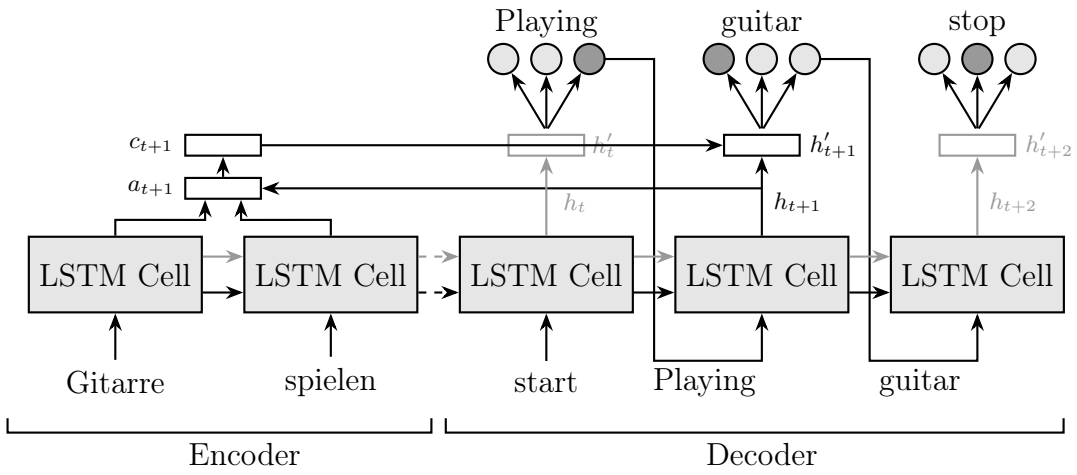


Abbildung 2.13: Encoder-Decoder-Modell mit Attention-Mechanismus gemäß Abbildung 2.7 [PC20] (S. 20.

Im Folgenden wird der Attention-Mechanismus basierend auf [PC20] (S. 19-20) sowie Abbildung 2.13 genauer erläutert.
Wie in Abbildung 2.13 zu sehen ist, wird ein sogenannter Alignment-Vektor a aus den Output-Sequenzen des Encoders gebildet. Dieser enthält Informationen darüber, welche Teile der Sequenz besonders berücksichtigt werden sollen. Aus dem Alignment-Vektor a und dem Output h des Decoders wird dann ein Kontext-Vektor c gebildet, der Informationen darüber enthält, welche Teile des Outputs h wie stark berücksichtigt werden sollen. Der finale Output h' des Decoders wird schließlich aus dem Kontextvektor und dem Output h berechnet.
Auf diese Weise kann der Decoder die gesamte Sequenz des Encoders berücksichtigen und sich insbesondere auf wichtige Stellen des Encoder-Outputs fokussieren, anstatt sich nur auf den Einbettungsvektor des Encoders zu konzentrieren.
Bis Mitte 2017 galten laut [PC20] (S. 21) rekurrente neuronale Netze (RNNs) als State-of-the-Art, bis schließlich Transformer-Modelle aufkamen. Transformer-Modelle benötigen keine Wiederholungen mehr. Stattdessen können ganze Sequenzen parallel verarbeitet werden, was sie wesentlich effizienter macht. In dieser Arbeit wird jedoch nicht näher auf Transformer eingegangen.

3

Methodik

Das Ziel dieser Arbeit ist es, ein großes Sprachmodell zur Bewertung von Klauseln einzusetzen. Auf Basis dieser Bewertungen wählt der Theorembeweiser gezielt die geeigneten Klauseln für die Beweisführung aus.
In jedem Schritt der Beweisführung stehen zahlreiche Klauseln zur Verfügung, die zum Bilden von Inferenzen genutzt werden können. Zur Bewertung dieser Klauseln werden bereits erfolgreiche Beweise analysiert. Das Ziel ist es, ein Ähnlichkeitsmaß zwischen Symbolen und Beweisaufgaben zu entwickeln, um die Klauseln gezielt für die Inferenzbildung auswählen zu können.
Das große Sprachmodell soll durch die Analyse bereits erfolgreich gelöster Beweise lernen, wie gut Symbole zu Beweisaufgaben passen. Auf Basis dieses trainierten Modells können dann Vorhersagen für zukünftige Beweisaufgaben getroffen werden, wie gut die Symbole zu einem zu beweisenden Ziel passen.
Die Klauseln werden vom Beweiser auf Grundlage dieser Symbolbewertungen beurteilt und entsprechend für die Inferenz ausgewählt. Dieser Ansatz könnte den Beweisprozess potenziell verbessern.
Der Methodikteil dieser Bachelorarbeit behandelt die Auswahl eines Beweisers. Zudem wird die Auswahl und das Fine-tuning des passenden Sprachmodells behandelt. Ein weiterer Punkt ist die Generierung der Daten zum Training und Test des Sprachmodells. Außerdem wird die Generierung der Daten für die Analyse der späteren Beweisführung durch den Beweiser behandelt. Die eigentliche Analyse und die Ergebnisse, die im Rahmen der Arbeit erzielt wurden, sind ebenfalls Teil des Methodikteils. Abschließend wird eine Diskussion dieser Ergebnisse durchgeführt.

3.1 Auswahl des Beweisers

Der Eprover ist gemäß [Sch21] (S. 6) ein automatischer Theorembeweiser, der auf dem Prinzip der Gleichheitslogik basiert und eine angepasste Version des Superpositionskalküls verwendet. Der Beweisprozess wird laut [Sch21] (S. 6) durch eine modifizierte Variante des Given-Clause Algorithmus gesteuert. Die Konfiguration des Beweisvorgangs ist äußerst flexibel, wie in [Sch21] (S. 17) beschrieben, und kann durch eine Vielzahl von Parametern sowie den Einsatz diverser Heuristiken

beeinflusst werden.
Besonders interessant für diese Arbeit ist die Möglichkeit, dem Eprover eine Heuristik in Form einer „Clause Evaluation Function“, bezeichnet als `<eval-fun>`, zuzuweisen, wie in [Sch21] (S. 17) erläutert. Die Funktion beeinflusst die Auswahl von Klauseln im Eprover, um neue Schlussfolgerungen zu ziehen und Inferenzen zu bilden. Dies ermöglicht eine gezielte Steuerung des Beweisprozesses.
Gemäß [Sch21] (S. 22) wird eine Heuristik wie folgt definiert:

$$(\texttt{<heu-element>}, \ldots, \texttt{<heu-element>}) \tag{3.1}$$

Hierbei ist jedes `<heu-element>` ein sogenanntes heuristisches Element, das laut [Sch21] (S. 22) wie folgt definiert ist:

$$\texttt{<int>*<eval-fun>} \tag{3.2}$$

Dabei gibt `<int>` einen ganzzahligen Wert an, der den Anteil der Klauseln bestimmt, die von der Heuristik bewertet werden sollen, und `<eval-fun>` steht für eine Evaluationsfunktion.
Ein Beispiel für eine Heuristik mit zwei heuristischen Elementen wäre:

Beispiel 3.1. (5*`<eval-fun>`, 1*`<eval-fun>`)

Dies bedeutet, dass von jeder Gruppe von sechs Klauseln fünfmal die erste und einmal die zweite Evaluationsfunktion angewendet wird.
Eine „Clause Evaluation Function“ (`<eval-fun>`) wird gemäß [Sch21] (S. 21) als eine Instanziierung einer „Generic Weight Function“ definiert.
Eine „Generic Weight Function“, ist gemäß [Sch21] (S. 20) eine Funktion, die einer Klausel einen Wert zuordnet, wobei Klauseln mit niedrigeren Werten bevorzugt ausgewählt werden. Eine solche Funktion besteht aus einer sogenannten „Priority Function“ (`<prio-fun>`) und einer Reihe von Parametern, die die „Generic Weight Function„ beeinflussen.
Die „Priority Function“ (`<prio-fun>`) identifiziert nach [Sch21] (S. 17) eine Teilmenge der Klauseln. Dies ist jedoch für Heuristiken laut [Sch21] (S. 17) weniger sinnvoll, da die Heuristik in der Regel auf alle Klauseln angewendet werden soll, die Auswahl einer echten Teilmenge von Klauseln ist eher zur Beeinflussung der Suchstrategie gedacht.
Aus diesem Grund wurde in dieser Arbeit lediglich die in [Sch21] (S. 20) definierte „Priority Function“ ConstPrio gewählt, die alle Klauseln als Teilmenge auswählt, was im Rahmen einer Heuristik sinnvoll ist.
Folgende „Generic Weight Functions“ wurden im Rahmen dieser Arbeit verwendet.
Die „FunWeight Funktion“ spielt eine zentrale Rolle in dieser Arbeit, da sie es ermöglicht, Funktionssymbolen bestimmte Werte zuzuweisen, die anschließend zur Bewertung von Klauseln verwendet werden.
Die Definition der „FunWeight Funktion“ lautet laut [Sch21] (S. 21) wie folgt:

$$\text{FunWeight}(\mathit{prio}, \mathit{fweight}, \mathit{vweight}, \mathit{termpen}, \mathit{litpen}, \mathit{posmult}, \mathit{fun} : \mathit{fweight}, \ldots)$$

Im Folgenden wird die Bedeutung der Parameter auf Basis von [Sch21] (S. 20-21) in Form einer Tabelle näher erläutert:

Parameter	**Bedeutung**
prio	Die oben beschriebene „Priority Function“.
fweight	Der Wert, mit dem Funktionssymbole belegt werden.
vweight	Der Wert, mit dem Variablen belegt werden.
termpen	Multiplikator für maximale Terme bezüglich der Literale nach der definierten Termordnung.
litpen	Multiplikator für maximale Literale nach der definierten Termordnung.
posmult	Multiplikator für positive Literale.
fun : *fweight*...	Eine Reihe von Gewichten für Funktionssymbole, die anstelle des Wertes, der in *fweight* definiert ist, verwendet werden.

Tabelle 3.1: Erklärung der Parameter für die „FunWeight Funktion“ gemäß [Sch21] (S. 20-21).

Durch die Parameter der „FunWeight Funktion“ werden den einzelnen Symbolen und Variablen in einer Klausel Werte zugewiesen. Diese Werte werden anschließend addiert, um eine finale Bewertung für die Klausel zu ermitteln.
Ein mögliches Beispiel für die „FunWeight Funktion“ wäre:

Beispiel 3.2. FunWeight(ConstPrio, 2, 1, 1, 1, 1, f:2, g:5, j:3, a:0, b:0)

Hierbei repräsentieren die Symbole f, g, j, a, b jeweilige Funktionssymbole.
Eine weitere „Generic Weight Function“, die in der Arbeit verwendet wurde, ist die „FIFOWeight Funktion“. Diese ist gemäß [Sch21] (S. 21) wie folgt definiert:

FIFOWeight(*prio*)

Hierbei ist *prio* eine „Priority Function“. Die „FIFOWeight Funktion“ ordnet die Klauseln nach dem FIFO-Prinzip (First-In-First-Out) und bewertet sie entsprechend, wobei zuletzt verarbeitete Klauseln einen niedrigeren Wert erhalten.
Ein Beispiel für die Verwendung der „FIFOWeight Funktion“ könnte wie folgt aussehen:

Beispiel 3.3. FIFOWeight(constPrio)

Gemäß [Sch21] (S. 22) kann eine Heuristik durch den Parameter `-x` an den Eprover mitgegeben werden. Ein Beispiel für den Aufruf des Eprovers mit einer solchen Heuristik wäre wie folgt:

Beispiel 3.4. ./eprover -x'(5*FunWeight(ConstPrio,2,1,1,1,1, f:2, g:5, j:3, a:0, b:0), 1*FIFOWeight(ConstPrio))' LUSK6.lop

Wobei „LUSK6.lop“ eine Datei ist, die das zu beweisende Ziel zusammen mit Axiomen enthält.
Dem Eprover kann gemäß [Sch21] (S. 6) ein Problem in Prädikatenlogik gegeben werden, welches dann vor dem Beweisvorgang in Klauselform umgewandelt wird.

Der Eprover unterstützt laut [Sch21] (S. 33) eine Bandbreite an Eingabeformaten. Insbesondere wird in dieser Arbeit die TPTP-Syntax verwendet, die gemäß [Sch21] (S. 34) vom Eprover unterstützt wird. Die TPTP-Syntax ist die standardisierte Eingabesyntax für Theorembeweiser, weshalb Eprover und auch Adimen-SUMO diese nutzen.
In Bezug auf die Ausgabeoptionen bietet Eprover nach [Sch21] (S. 38) eine vielfältige Auswahl an Formaten sowie, laut [Sch21] (S. 36), zahlreiche Möglichkeiten zur Anpassung der Ausgabe über Output-Levels. Gemäß [Sch21] (S. 38) kann ein „prove-object“ generiert werden, das detaillierte Informationen über den Beweisvorgang enthält. Zusätzlich können nach [Sch21] (S. 40-41) Ressourceninformationen und spezifische Details zum Beweisprozess ausgegeben werden, wie beispielsweise die Anzahl der Schritte im Saturationsalgorithmus, wie im Abschnitt 2.1.6 beschrieben, die für das Erreichen des Beweises erforderlich sind, sowie Angaben zu den verarbeiteten Klauseln und anderen relevanten Aspekten.
Dies ist insbesondere für die Analyse des Beweisprozesses im Rahmen der Arbeit interessant.
Eprover wurde im Rahmen dieser Arbeit verwendet, da er nach [Sch21] (S. 3) in der wissenschaftlichen Gemeinschaft etabliert ist und mehrfach ausgezeichnet wurde laut [epr]. Außerdem ist er in der Anleitung [Sch21] sehr gut dokumentiert. Ein weiterer Vorteil des Eprovers besteht darin, dass er in C programmiert wurde, was ihn besonders schnell macht. Darüber hinaus bietet er eine hohe Anpassbarkeit des Beweisvorgangs sowie der Ein- und Ausgabe, wie bereits erwähnt.

3.2 Auswahl des Sprachmodells

Das Ziel ist es, mithilfe eines Large Language Models einen Wert für die semantische Ähnlichkeit zwischen einem zu beweisenden Ziel und einem Symbol zu berechnen. Auf dieser Grundlage kann dann eine Vorhersage getroffen werden, ob ein Symbol gut oder weniger gut zu einer Beweisaufgabe passt. Die Werte für die Symbolnamen können anschließend Eprover übergeben werden, der dann basierend darauf die Klauseln gezielt auswählen kann.
Ein Ansatz, wie in [BGMIM20] (S. 232) beschrieben, besteht darin, ein Modell auszuwählen, das einen Einbettungsvektor für einen Text berechnet. Anschließend kann, gemäß [BGMIM20] (S. 236), die Kosinusähnlichkeit verwendet werden, um die Einbettungsvektoren verschiedener Texte miteinander zu vergleichen.
Die Kosinusähnlichkeit zwischen zwei Vektoren $\vec{v}$ und $\vec{w}$ ist gemäß [JM23] (S. 11) wie folgt definiert:

$$\text{cosine}(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|}$$

Die Kosinusähnlichkeit berechnet somit den Winkel zwischen den beiden Vektoren. Die Einbettungsvektoren ähnlicher Texte haben demnach einen Wert, der nahe bei Eins liegt. Null bedeutet, dass die Texte neutral sind, und minus Eins bedeutet, dass die Texte sehr verschieden voneinander sind.

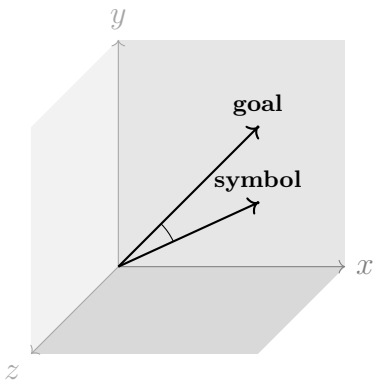


Abbildung 3.1: Kosinusähnlichkeit zwischen Ziel- und Symbol-Embedding in einem dreidimensionalen Vektorraum basierend auf [JM23] (S. 11).

Ein großes Sprachmodell, das Einbettungsvektoren berechnet, welche mittels der Kosinusähnlichkeit verglichen werden können, ist das „`all-MiniLM-L6-v2`“. Gemäß [all] bildet dieses Modell einen 256 Zeichen langen Text auf einen 384-dimensionalen Vektorraum ab, codiert dessen semantische Bedeutung und kann beispielsweise für Vergleiche verwendet werden. Das Modell besteht aus 22,7 Millionen Parametern und wurde mit einer Milliarden Trainingsdaten trainiert. Es wurde während der von Hugging Face organisierten Community-Week using JAX/Flax for NLP & CV entwickelt.
Es wurde für diese Arbeit gewählt, da es den oben genannten Anforderungen entspricht, gut dokumentiert ist und eines der am meisten heruntergeladenen Modelle auf Hugging Face ist, um die semantische Bedeutung von Texten zu vergleichen.
Das Modell wurde feinabgestimmt, wie in Kapitel 3.5 beschrieben, um die Ähnlichkeit zwischen einem zu beweisenden Ziel und einem Symbolnamen zu bestimmen.

3.3 Auswahl der Wissensbasis

Um das Sprachmodell zu trainieren und für spätere Analysen werden Daten benötigt. Hierfür wird eine Wissensbasis benötigt, aus der die Daten generiert werden können. Im Rahmen dieser Arbeit wurde eine Ontologie für die Wissensbasis verwendet.
Laut [GOS09] (S. 1) beschäftigt sich die Ontologie philosophisch gesehen mit der Natur und Struktur der Realität.
In der Informatik ist eine Ontologie gemäß [GOS09] (S. 2) ein Informationsobjekt, das darauf abzielt, formal die Struktur eines Systems zu beschreiben, indem alle relevanten Entitäten und Relationen zwischen den Entitäten basierend auf Beobachtungen festgelegt werden.
Gemäß [ALR14] (S. 80) ist es wichtig, dass eine Ontologie in einer formalen Sprache beschrieben wird. Dabei entsteht ein Kompromiss zwischen der Ausdrucksstärke der formalen Sprache und ihrer effizienten Berechenbarkeit.
In dieser Arbeit wurde die Adimen-SUMO-Ontologie verwendet. Adimen-SUMO

basiert laut [ALR14] (S. 81) auf SUMO (Suggested Upper Merged Ontology) [NP01].
Eine Upper Ontology ist laut [ALR14] (S. 83) eine Ontologie, die sich mit Konzepten, Relationen und Axiomen beschäftigt, die generisch und abstrakt sind. Allerdings bieten Upper Ontologies eine Struktur, um eine domänenspezifische Ontologie zu erstellen.
Ein Problem von SUMO ist gemäß [ALR14] (S. 83-84), dass die Ontologie in SUMO-KIF, einem Dialekt von KIF (Knowledge Interchange Format), vorliegt. Das Problem von SUMO-KIF ist laut [ALR14] (S. 84), dass SUMO-KIF die Beschreibung von Wissen in Form vieler unterschiedlicher Logiken erlaubt und nicht ausschließlich in Prädikatenlogik ist. Dadurch ist es gemäß [ALR14] (S. 81) schwierig, große Ontologien wie SUMO mit Beweisern zu verarbeiten.
Um dieses Problem zu lösen, wurde gemäß [ALR14] (S. 80) Adimen-SUMO entwickelt. Es handelt sich dabei um eine Ontologie, in der 88 Prozent der Axiome von SUMO erfolgreich in Prädikatenlogik erster Stufe übersetzt wurden.
Dies hat den Vorteil, dass laut [ALR14] (S. 81) Theorembeweiser für Prädikatenlogik erster Stufe wie der Eprover verwendet werden können, der sehr effizient ist, um Aussagen in SUMO zu beweisen.
Adimen-SUMO wurde gemäß [ALR14] (S. 95) in TPTP-Syntax übersetzt, welche im Abschnitt 2.1.2 näher erläutert wird.
Adimen-SUMO kann als Ontologie zusammen mit Beweisaufgaben heruntergeladen werden und ist in Form einer Ordnerstruktur organisiert.
Zwei Ordner sind für diese Arbeit besonders relevant. Der Ordner „axioms“ enthält die aus SUMO übersetzten Axiome in TPTP-Syntax. Der Ordner „goals“ enthält eine Sammlung von Zielen (goals) bzw. Benchmarks, die anhand des Dateinamens in verschiedene Kategorien eingeteilt sind. Diese Ziele liegen in Form von „Conjectures“ in der TPTP-Syntax vor.
Da der Eprover Probleme im TPTP-Format verarbeiten kann, ist es einfach, Beweise zu führen und aus diesen Daten Material für Training und Analyse zu generieren.
Um mehr Daten zu erhalten, wurden auch Beweisprobleme aus dem Projekt „Applying Closed World Assumption“ (CWA), das näher in [ÁGDR18] beschrieben wird, genutzt. Das Projekt enthält weitere Beweisprobleme, die als zusätzliche Daten verwendet wurden.
Für diese Arbeit wurde Adimen-SUMO gewählt, da es zahlreiche Beweisaufgaben beinhaltet, die in einer Ontologie untersucht werden können. Die Ziele und Axiome sind in der TPTP-Syntax in Prädikatenlogik erster Stufe formuliert.
Dadurch kann der in dieser Arbeit verwendete Eprover eingesetzt werden, um zu versuchen die Ziele zu beweisen.
Ein weiterer Grund für die Wahl von Adimen-SUMO ist, dass die Symbolnamen den Wörtern der natürlichen Sprache sehr ähnlich sind. Dies ist wichtig, da ein Sprachmodell später die semantische Ähnlichkeit zwischen Symbolnamen und den zu beweisenden Zielen bewerten soll.
Somit können Daten für das Training und die Analyse generiert werden.

3.4 Generierung von Trainings-, Test- und Analyse-Daten

Im Folgenden wird beschrieben, wie die Trainings- und Testdaten für das Finetuning des Sprachmodells sowie Daten für spätere Analysen generiert wurden.
Die Analyse-Daten dienen dazu, zu evaluieren, wie gut das trainierte neuronale Netz mittels des Eprovers performt. Ziel ist es, ein Sprachmodell zu entwickeln, das beurteilt, wie gut ein logisches Symbol aus einer Klausel (Funktions-, Prädikaten- oder Konstantensymbol) zu einer gegebenen Beweisaufgabe passt. Basierend darauf werden die Klauseln bewertet und entsprechend vom Given-Clause Algorithmus ausgewählt.
Um die Beschreibung dieser Daten klarer und verständlicher zu machen, werden in diesem Abschnitt und im weiteren Verlauf der Arbeit die folgenden Begriffe verwendet. Diese Begriffe dienen dazu, die Bezeichnungen von Daten im Zusammenhang mit einem Beweis, der mithilfe eines Beweisers durchgeführt wurde, zu vereinfachen:

- **Positive Klauseln** (synonym: *positiv bewertete Klauseln*): Klauseln, die vom Beweiser verarbeitet wurden und im Beweis vorkommen.
- **Negative Klauseln** (synonym: *negativ bewertete Klauseln*): Klauseln, die vom Beweiser verarbeitet wurden, aber nicht im Beweis vorkommen.
- **Positive Symbole** (synonym: *positiv bewertete Symbole*): Symbole, die in mindestens einer positiven Klausel vorkommen.
- **Negative Symbole** (synonym: *negativ bewertete Symbole*): Symbole, die in mindestens einer negativen Klausel, aber in keiner positiven Klausel vorkommen.
- **Neutrale Symbole** (synonym: *neutral bewertete Symbole*): Symbole, die weder in positiven noch in negativen Klauseln vorkommen und somit im Kontext des Beweisvorgangs vom Beweiser nicht verwendet wurden.

Ein Trainingsdatum besteht aus einer Beweisaufgabe, einem Symbol und einem Score, der angibt, wie gut das Symbol zur Beweisaufgabe passt. Die Trainingsdaten sind somit ein 3-Tupel, bestehend aus $(\text{Beweisaufgabe}, \text{Symbol}, \text{Score})$. Die Trainings-, Test- und Analyse-Daten wurden unter Verwendung von Adimen-SUMO 2.6 und dem Applying CWA Projekt generiert.
Die Beweisaufgaben im Goals-Ordner von Adimen-SUMO und Applying CWA lassen sich entsprechend ihrer Namen in Kategorien sortieren.
Nachfolgend ist das Ergebnis der Kategorisierung der Beweisaufgaben zu sehen:

Kategorie	**Anzahl der Beweisaufgaben**
whiteBoxTruthTest	8010
whiteBoxFalsityTest	8010
negatedAntonymPattern	3004
antonymPattern	3004
negatedAgentRelation	829
negatedMultipleMapping	151
resultRelation	788
negatedCommonSubclassEvent	2011
commonSubclassEvent	2011
negatedResultRelation	788
instrumentRelation	348
negatedSubclassEvent	350
negatedInstrumentRelation	348
agentRelation	829
multipleMapping	151
equalEvent	24
subclassEvent	350
negatedEqualEvent	24
goal	64

Tabelle 3.2: Ergebnis der Kategorisierung der Beweisaufgaben von Adimen-SUMO.

Kategorie	**Anzahl der Beweisaufgaben**
negatedNounHyponymPattern	268
verbHyponymPattern	9268
negatedAntonymPattern	2039
antonymPattern	3017
negatedVerbHyponymPattern	3017

Tabelle 3.3: Ergebnis der Kategorisierung der Beweisaufgaben von Applying CWA.

Um die Beweisaufgaben mithilfe von Eprover zu lösen, ist eine Wissensbasis in Form von Axiomen erforderlich. Diese Axiome sind in in der Datei „adimen.sumo.tstp“ im entsprechenden Axiom-Ordner von Adimen-SUMO und Applying CWA zu finden. Jede Beweisaufgabe wurde zusammen mit den Axiomen in einer Datei gespeichert, um sie mit dem Eprover lösen zu können. Anschließend wurden die Beweisaufgaben mithilfe des Eprovers versucht zu lösen, indem der folgende Befehl verwendet wurde:

```
eprover „Pfad zum Ziel mit Axiomen“ -auto -silent
-training-examples=3 -proof-object -cpu-limit=15 -R
-outfile=„Pfad zum Speichern des Outputs“.
```

Nachfolgend wird eine kurze Erläuterung der Optionen gegeben. Eine umfassende Beschreibung kann durch den Befehl `eprover -h` abgerufen werden.

Option	**Erklärung**
–auto	Automatischer Modus des Eprover, der versucht, Beweise automatisch zu finden, indem verschiedene Suchstrategien und Heuristiken angewendet werden. Darüber hinaus nutzt der Eprover die Selektionstechnik SInE, um die Menge der Axiome zu reduzieren.
–silent	Stiller Modus, bei dem der Eprover keine zusätzliche Ausgabe generiert, außer dem Beweis falls dieser gefunden wurde.
–training-examples=3	Klauseln, die nicht im Beweis enthalten sind, werden für das Training mit „trainneg“ markiert, während Klauseln, die im Beweis enthalten sind, mit „trainpos“ markiert werden.
–proof-object	Generiert ein Objekt, das den Beweis repräsentiert, falls einer gefunden wurde.
–cpu-limit=15	Begrenzt die CPU-Zeit in Sekunden, die Eprover für die Suche nach einem Beweis verwenden kann.
-R	Gibt die verwendeten Ressourcen aus.
–outfile	Spezifiziert den Pfad zur Ausgabedatei, in welcher die Ausgabe des Eprovers gespeichert wird.

Tabelle 3.4: Erläuterung der Aufrufoptionen des Eprovers.

Die Ausgabe des Eprovers wurde in zwei Kategorien unterteilt: Beweise, die erbracht werden konnten, und Beweise, die nicht erbracht werden konnten. Die Beweisziele der nicht erbrachten Beweise wurden für spätere Analysezwecke aufbewahrt, während die erbrachten Beweise für das Training des neuronalen Netzes verwendet wurden.
In der folgenden Tabelle sind die Anzahl der erbrachten und nicht erbrachten Beweise für jede Kategorie aufgeführt.

Kategorie	**Anzahl erbrachter Beweise**	**Anzahl nicht erbrachter Beweise**
agentRelation	4	825
antonymPattern	227	2777
commonSubclassEvent	580	1431
equalEvent	0	24
goal	22	39
instrumentRelation	2	346
multipleMapping	0	151
negatedAgentRelation	1	828
negatedAntonymPattern	23	2981
negatedCommonSubclassEvent	0	2011
negatedEqualEvent	2	22
negatedInstrumentRelation	0	348
negatedMultipleMapping	1	150
negatedResultRelation	3	785
negatedSubclassEvent	0	350
resultRelation	10	778
subclassEvent	83	267
whiteBoxFalsityTest	0	8010
whiteBoxTruthTest	3917	4093

Tabelle 3.5: Anzahl der erbrachten und nicht erbrachten Beweise nach Kategorien von Adimen-SUMO.

Kategorie	**Anzahl erbrachter Beweise**	**Anzahl nicht erbrachter Beweise**
antonymPattern	303	2714
negatedAntonymPattern	27	2990
negatedNounHyponymPattern	370	8898
negatedVerbHyponymPattern	24	2015
nounHyponymPattern	3209	6059
verbHyponymPattern	391	1648

Tabelle 3.6: Anzahl der erbrachten und nicht erbrachten Beweise nach Kategorien von Applying CWA.

Aus den Tabellen 3.5 und 3.6 ist ersichtlich, dass nur ein Bruchteil aller Beweisprobleme gelöst werden konnte. Von insgesamt 31073 Beweisproblemen in Adimen-SUMO wurden 4857 erfolgreich gelöst, während von 28648 Beweisproblemen in Applying CWA 4324 erfolgreich gelöst wurden. Insgesamt konnten 9199 Beweisprobleme von 59721 erfolgreich gelöst werden.
Die vorhandenen Beweise wurden aufgeteilt, wobei 80 Prozent der Beweise für das Training des Sprachmodells verwendet wurden und die Beweisziele der verbleiben-

den 20 Prozent für die spätere Analyse des Sprachmodells genutzt wurden. Dabei wurden Kategorien, in denen keine erbrachten Beweise vorliegen, vernachlässigt.

Kategorie	**Anzahl Beweise für das Training**	**Anzahl Beweise für die Analyse**
agentRelation	3	1
antonymPattern	181	46
commonSubclassEvent	463	117
goal	17	5
instrumentRelation	1	1
negatedAgentRelation	0	1
negatedAntonymPattern	18	5
negatedEqualEvent	1	1
negatedMultipleMapping	0	1
negatedResultRelation	2	1
resultRelation	7	3
subclassEvent	66	17
whiteBoxTruthTest	3133	784

Tabelle 3.7: Anzahl der erbrachten Beweise nach Kategorien für das Training und die Analyse des Sprachmodells von Adimen-SUMO.

Kategorie	**Anzahl Beweise für das Training**	**Anzahl Beweise für die Analyse**
antonymPattern	241	61
negatedAntonymPattern	21	6
negatedNounHyponymPattern	295	75
negatedVerbHyponymPattern	19	5
nounHyponymPattern	2567	642
verbHyponymPattern	312	79

Tabelle 3.8: Anzahl der erbrachten Beweise nach Kategorien für das Training und die Analyse des Sprachmodells von Applying CWA.

Die Tabellen 3.7 und 3.8 zeigen, dass von Adimen-SUMO insgesamt 3892 erbrachte Beweise für das Training und 983 erbrachte Beweise für die Analyse verwendet wurden. Von Applying CWA wurden 3455 erbrachte Beweise für das Training und 868 erbrachte Beweise für die Analyse genutzt. Zusammen wurden somit 7347 erbrachte Beweise für das Training und 1851 erbrachte Beweise für die spätere Analyse verwendet.
Um sowohl positive Trainingsdaten zu erzeugen als auch negative, wurden die positiven Klauseln (Klauseln, die in dem Beweis vorkommen), und die negativen Klauseln (Klauseln, die nicht in den Beweisen vorkommen), voneinander getrennt.

Hierbei wurde die Tatsache ausgenutzt, dass die Option `-training-examples=3` negative Klauseln mit „trainneg“ und positive Klauseln mit „trainpos“ markiert. Damit beurteilt werden kann, wie gut ein Symbol zu einer Beweisaufgabe passt, mittels eines großen Sprachmodells, ist es erforderlich, die Beweisaufgabe, die in Prädikatenlogik vorliegt, in natürlicher Sprache zu verbalisieren.
Zum Verbalisieren der Beweisaufgaben und zum Extrahieren von Symbolnamen aus den Klauseln war die Entwicklung eines Parsers für die TPTP-Syntax erforderlich.
Für diesen Zweck wurde ein Parse-Baum generiert, angelehnt an [NSS20]. Die folgenden Grammatikregeln wurden basierend auf [tpt] entwickelt und für den Top-Down-Parser verwendet:

- ⟨formula⟩ ::= (⟨formula⟩) | ⟨formula⟩ => ⟨formula⟩ |
  ⟨formula⟩ <=> ⟨formula⟩ | ⟨formula⟩|⟨formula⟩ | ⟨formula⟩&⟨formula⟩ |
  ![⟨variable⟩, . . . , ⟨variable⟩]⟨formula⟩ | ?[⟨variable⟩, . . . , ⟨variable⟩]⟨formula⟩ |
  ∼ ⟨formula⟩ | ⟨predicate⟩ | ⟨formula⟩
- ⟨predicate⟩ ::= ⟨atomic⟩(⟨term⟩, . . . , ⟨term⟩) | ⟨term⟩=⟨term⟩ | ⟨term⟩!=⟨term⟩ |
  ⟨atomic⟩
- ⟨term⟩ ::= ⟨variable⟩ | ⟨function⟩(⟨term⟩, . . . , ⟨term⟩) | ⟨constant⟩

Beispiel 3.5. Seien A, B und D nullstellige Prädikate, p ein einstelliges Prädikat und c eine Konstante.
Die Formel (A => B)|(p(c)&D) könnte beispielsweise mit folgenden Grammatikregeln geparst werden:

Grammatikregeln	**Zeichenketten**
⟨formula⟩ ::= ⟨formula⟩ | ⟨formula⟩	(A => B)|(p(c)&D)
⟨formula⟩ ::= (⟨formula⟩)	(A => B)
⟨formula⟩ ::= ⟨formula⟩ => ⟨formula⟩	A => B
⟨formula⟩ ::= ⟨predicate⟩	A
⟨predicate⟩ ::= ⟨atomic⟩	A
⟨formula⟩ ::= ⟨predicate⟩	B
⟨predicate⟩ ::= ⟨atomic⟩	B
⟨formula⟩ ::= (⟨formula⟩)	(p(c)&D)
⟨formula⟩ ::= ⟨predicate⟩	D
⟨predicate⟩ ::= ⟨atomic⟩	D
⟨formula⟩ ::= ⟨formula⟩&⟨predicate⟩	p(c)&D
⟨formula⟩ ::= ⟨predicate⟩	p(c)
⟨predicate⟩ ::= ⟨atomic⟩(⟨term⟩, . . . ,⟨term⟩)	p(c)
⟨term⟩ ::= ⟨constant⟩	c
⟨formula⟩ ::= ⟨predicate⟩	D
⟨predicate⟩ ::= ⟨atomic⟩	D

Tabelle 3.9: Beispiel für das Parsen einer Formel in TPTP-Syntax mit den entsprechenden verwendeten Grammatikregeln.

Ein möglicher Parse-Baum für die Formel (A => B)|(p(c)&D) sieht wie folgt aus:

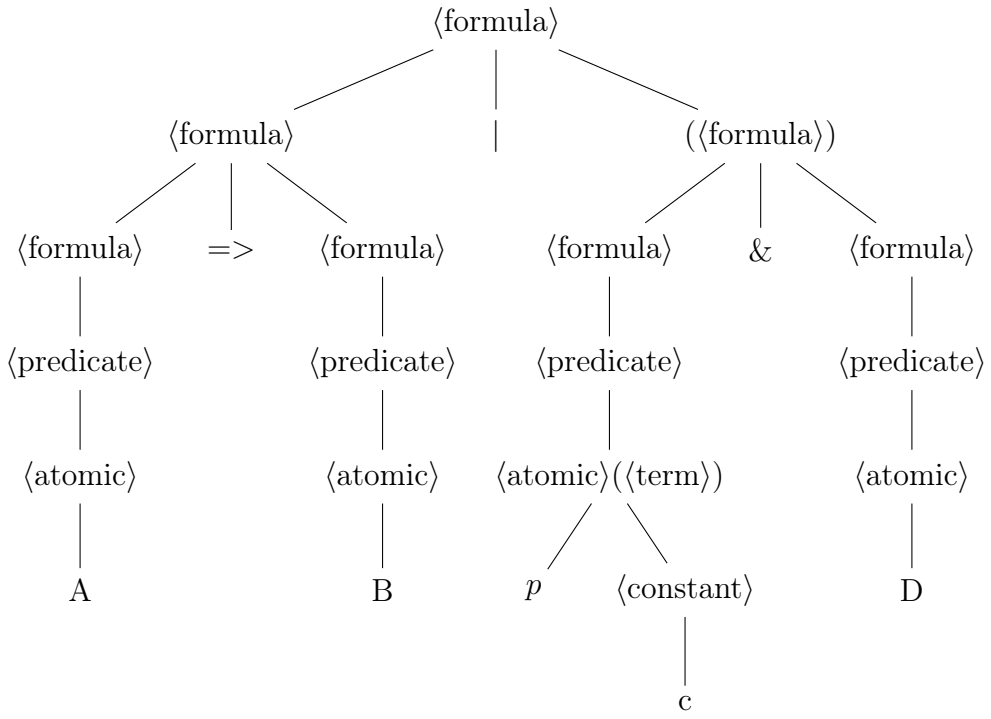


Abbildung 3.2: Beispiel für einen Parse-Baum einer Formel in TPTP-Syntax.

Zur Verbalisierung der Beweisaufgaben wurde der Parsebaum rekursiv durchlaufen und die syntaktischen Elemente wurden in Anlehnung an [MH07] wie folgt in natürliche Sprache übersetzt:

Syntaktische Elemente	**Natürliche Sprache**
&	and
|	or
$\sim$	not
$=$	is equal to
$!=$	is not equal to
$\Leftrightarrow$	is equivalent to
$\Rightarrow$	implies that
$![x_1, \ldots, x_n]$	for all x_1 and ... and x_n
$?[x_1, \ldots, x_n]$	there exists at least one x_1 and ... and x_n
true	is true
false	is false
prdicateName	predicateName
prdicateName(x_1)	x_1 predicateName
prdicateName(x_1, x_2)	x_1 predicateName x_2
prdicateName($x_1, \ldots, x_n$)	predicateName of x_1 and ... and x_n
constantName	constantName
functionName(x_1)	x_1 functionName
functionName(x_1, x_2)	x_1 functionName x_2
functionName($x_1, \ldots, x_n$)	functionName of x_1 and ... and x_n

Tabelle 3.10: Übersetzung der syntaktischen Elemente in natürliche Sprache.

Die Position eines Terms in einem Satz variiert je nach Stelligkeit des Prädikats oder der Funktion. Dies liegt daran, dass solche Muster häufig im Satzbau vorkommen. Ein Beispiel hierfür ist das zweistellige Prädikat walksTo(x, y), das als x walks to y übersetzt wird.
Für die Übersetzung von Symbolnamen in natürliche Sprache wurde das in [Sch22] beschriebene Mapping verwendet. Dieses bildet 3179 Symbolnamen auf 2561 Wörter und Sätze in natürlicher Sprache ab. Die unterschiedlichen Zahlen ergeben sich daraus, dass manche logischen Symbole auf dasselbe Wort abgebildet bzw. mit diesem übersetzt werden.
Ein Problem bestand darin, dass einige Wörter in natürlicher Sprache teilweise mit Unterstrichen geschrieben sind oder im Camel Case vorliegen.
Die Symbolnamen wurden aus diesem Grund sowohl an den Unterstrichen als auch an den Stellen, an denen sich die Groß- und Kleinschreibung ändert (Camel Case), getrennt. Dies geschieht, da diese häufig benutzt werden, um zusammengesetzte komplexe Zusammenhänge auszudrücken. Zum Beispiel wird der Symbolname „isSuny“ zu „is Sunny“ aufgeteilt.

Beispiel 3.6. Die Formel isSunny(x)& $\sim$ isRaining(x) => goodDay(x) in TPTP-Syntax (siehe Kapitel 2.1.2) wird somit wie folgt in natürliche Sprache übersetzt: x is sunny and not x is raining implies that x good day.

Die Übersetzung ist nicht besonders präzise, jedoch besteht die Hoffnung, dass die syntaktische Struktur der Übersetzung ausreicht, um vorherzusagen, wie gut ein

Symbol zu der übersetzten Beweisaufgabe passt.
Ebenfalls müssen die Symbolnamen aus den Klauseln extrahiert und übersetzt werden.
Zur Extraktion und Übersetzung der Symbolnamen aus den Klauseln wurde ebenfalls der Parsebaum rekursiv durchlaufen.
Dabei wurden die einzelnen Symbolnamen (Prädikatensymbole, Funktionsymbole, Konstantensymbole) aus den positiven und negativen Klauseln extrahiert.
Die einzelnen Symbolnamen wurden, wie bereits oben beschrieben, mithilfe der Datei, welche die Symbolnamen auf natürliche Sprache abbildet, übersetzt.
Für jedes der 2561 Wörter und Sätze aus der Datei, die potenzielle Übersetzungen für die Symbolnamen darstellen, wurde ein Wert berechnet. Dieser Wert gibt an, wie gut der übersetzte Symbolname zu einer übersetzten Beweisaufgabe passt.
Bei der Berechnung der Werte wurden folgende Unterscheidungen getroffen:

1. Übersetze Symbolnamen, die in einem Beweis vorkommen (Symbole aus positiven Klauseln).
2. Übersetze Symbolnamen, die vom Beweiser verarbeitet wurden, jedoch nicht in einem Beweis auftreten (Symbole aus negativen Klauseln, die in keiner positiven Klausel vorkommen).
3. Potenzielle Übersetzungen für Symbolnamen, die weder im Beweis auftreten noch vom Beweiser verarbeitet wurden (Symbole weder aus positiven noch aus negativen Klauseln).

Symbolnamen, die in einem Beweis auftauchen, wurden positiv bewertet, da sie vom Beweiser verwendet wurden und einen Beitrag zum Beweis geleistet haben.
Die Relevanz der übersetzten Symbolnamen, die in einem Beweis auftreten, wurde jeweils anhand ihres sogenannten TF-IDF-Werts bestimmt.
Der TF-IDF-Wert ist eine Heuristik, die die Relevanz eines Begriffs für ein Dokument angibt. Laut [Rob04] (S. 503) ist TF-IDF ein Gewichtungsschema gemäß der Struktur $\mathrm{TF} * \mathrm{IDF}$.
Es gibt verschiedene Ansätze, um den IDF-Wert und den TF-Wert zu berechnen.
IDF steht für inverse Dokumentenfrequenz (Inverse Document Frequency). Die zugrunde liegende Idee laut [Rob04] (S. 503) ist, dass ein Begriff, der in vielen Dokumenten vorkommt, weniger Gewicht erhalten sollte als ein Begriff, der in nur wenigen Dokumenten vorkommt.
Gemäß [Rob04] (S. 504) kann der IDF-Wert wie folgt definiert werden:

$$\mathrm{idf}(t_i) = \log_{10}(\frac{N}{n_i})$$

Hierbei steht t_i für den Term t_i, N für die Anzahl der Dokumente und n_i für die Anzahl der Dokumente, in denen t_i vorkommt.
Nach [Rob04] (S. 506) kann der IDF-Wert auch als die Menge an Information interpretiert werden, die der Begriff t_i trägt, basierend auf dem Shannon-Informationstheorem.
DF steht für Dokumentenhäufigkeit (Document Frequency). Vereinfacht gesagt gibt die Dokumenthäufigkeit an, wie oft ein Begriff in einem Dokument vorkommt.

Laut [Rob04] (S. 503) deutet ein höherer Wert darauf hin, dass der Begriff für das Dokument wichtiger ist.
Im Gegensatz zur Definition in [Rob04] (S. 513) wurde in dieser Arbeit der TF-Wert gemäß [JM23] (S. 12) festgelegt:

$$\mathrm{tf}(t_i, d_j) = \begin{cases} 1 + \log_{10}(\mathrm{count}(t_i, d_j)) & \text{für } \mathrm{count}(t_i, d_j) > 0 \\ 0 & \text{sonst} \end{cases}$$

Hierbei steht t_i für den Term t_i, d_j für das Dokument d_j und $\mathrm{count}(t_i, d_j)$ für die Anzahl der Vorkommen des Terms t_i im Dokument d_j.
Die vollständige Formel für den TF-IDF-Wert lautet wie folgt:

$$\mathrm{tfidf}(t_i, d_i) = \mathrm{tf}(t_i, d_j) * \mathrm{idf}(t_i)$$

Der TF-IDF-Wert wurde wie bereits erwähnt verwendet, um die Relevanz eines übersetzten Symbolnamens aus einem Beweis (Symbole aus positiven Klauseln) für eine Beweisaufgabe zu bestimmen. Alle übersetzten Symbolnamen eines Beweises wurden als ein Dokument betrachtet. Zur Berechnung der TF-IDF-Werte wurden dann alle Beweise aus den Trainingsdaten herangezogen.
Auf diese Weise kann der Beitrag eines Symbols bzw. die Bedeutung eines Symbols für den entsprechenden Beweis ermittelt werden.

Beispiel 3.7. Angenommen, es liegen zwei Beweisaufgaben G_1 und G_2 vor, jeweils mit den entsprechend übersetzten und extrahierten Symbolnamen, die im Beweis vorkommen.

```
G_1:{
    "for all humans humans drink water": {
        "positiveSymbols": ["wine", "wine"],
    }
}
G_2:{
    "there exists at least one human human eats apples": {
        "positiveSymbols": ["wine", "meal"],
    }
}
```

Listing 3.1: Beweisaufgaben mit extrahierten übersetzten Symbolnamen.

Dadurch können die folgenden TF-IDF-Werte für die Symbole berechnet werden:
Für „for all humans humans drink water“:

$$\mathrm{tfidf}(\mathrm{wine}, G_1) = (1 + \log_{10}(1 + 1)) * \log_{10}(\frac{2}{2}) = 0$$

Für „there exists at least one human human eats apples“:

$$\mathrm{tfidf}(\mathrm{wine}, G_2) = (1 + \log_{10}(1)) * \log_{10}(\frac{2}{2}) = 0$$

$$\mathrm{tfidf}(\mathrm{meal}, G_2) = (1 + \log_{10}(1)) * \log_{10}(\frac{2}{1}) = \log_{10}(2)$$

Die nachfolgenden Trainingsdaten können für die beiden Beweisaufgaben G_1 und G_2 erstellt werden:

(„for all humans humans drink water“, „wine“, 0)
(„there exists at least one human human eats apples“, „wine“, 0)
(„there exists at least one human human eats apples“, „meal“, log(2))

Symbolnamen, die nicht im Beweis auftauchen, jedoch vom Beweiser verarbeitet wurden, wurden negativ bewertet. Dies liegt daran, dass sie vom Beweiser verarbeitet wurden, aber keinen Beitrag zum Beweis geleistet haben. Aus diesem Grund sollte die Verarbeitung der Symbole im Beweiser reduziert werden.
Ein Wert dafür, wie negativ ein Symbolname zu einer Beweisaufgabe zu bewerten ist, wurde anhand der Häufigkeit berechnet. Symbole, die häufiger vom Beweiser verarbeitet wurden, ohne einen Beitrag zum Beweis zu leisten, sind negativer zu bewerten als Symbole, die weniger oft verarbeitet wurden.

Beispiel 3.8. Angenommen, folgende Beweisaufgabe G_1 ist gegeben, zusammen mit den entsprechenden übersetzten extrahierten Symbolnamen, die vom Beweiser verwendet wurden, aber nicht im Beweis vorkommen.

```
1 G_1:{
2     "for all humans humans drink water": {
3         "negativeSymbols": ["car", "car", "chair"]
4     }
5 }
```

Listing 3.2: Beweisaufgabe mit extrahierten übersetzten Symbolnamen.

Die nachfolgenden Trainingsdaten können für die Beweisaufgabe G_1 erstellt werden:

(„for all humans humans drink water“, „car“, 2)
(„for all humans humans drink water“, „chair“, 1)

Es ist schwierig, die Relevanz der Symbolnamen für die Beweisaufgabe zu beurteilen, wenn sie nicht im Beweis auftauchen und vom Beweiser nicht verwendet wurden. Aus diesem Grund wurden sie neutral bewertet und erhalten den Wert null.

Beispiel 3.9. Angenommen, folgende Beweisaufgabe G_1 ist gegeben, mit den entsprechenden übersetzten extrahierten Symbolnamen und das Wort „cooking“, das nicht im Beweis enthalten ist und nicht vom Beweiser verwendet wurde:

```
1 G_1:{
2     "for all humans humans drink water": {
3         "positiveSymbols": ["wine", "wine"],
4         "negativeSymbols": ["car", "car", "chair"]
5     }
6 }
```

Listing 3.3: Beweisaufgabe mit extrahierten übersetzten Symbolnamen.

Das folgende Trainingsdatum für das Wort „cooking“ kann für die Beweisaufgabe G_1 generiert werden:

(„for all humans humans drink water“, „cooking“, 0)

Wie bereits in Abschnitt 3.2 erwähnt, wird für den Vergleich der Embeddings die Kosinusähnlichkeit verwendet, die auf den reellen Zahlen definiert ist und Werte im Intervall von 1 bis -1 annimmt. Ein Wert von 1 bedeutet dabei, dass ein Symbol sehr gut zu einem zu beweisenden Ziel passt, während ein Wert von -1 bedeutet, dass ein Symbol sehr schlecht zu einem zu beweisenden Ziel passt.
Es ist jedoch nicht garantiert, dass der TF-IDF-Wert im Bereich von 1 und 0 liegt und dass der berechnete Häufigkeitswert der Symbole, die nicht im Beweis vorkommen, aber vom Beweiser verarbeitet wurden, im Bereich von 0 und -1 liegt.
Aus diesem Grund wurde die sogenannte Min-Max-Normalisierung verwendet, um die Daten zwischen 1 und 0 bzw. 0 und -1 zu normalisieren. Die Min-Max-Normalisierung ist eine Methode, um Daten in einem bestimmten Intervall zu normalisieren. Die zugrunde liegende Idee ist recht einfach: Durch eine lineare Transformation werden die Werte eines Datensatzes so angepasst, dass sie innerhalb eines definierten Intervalls liegen.
Die Min-Max-Normalisierung wurde in der Arbeit basierend auf [SZK06] (S. 735) wie folgt definiert:

$$g(x) = \frac{(x - A)}{(B - A)} \cdot (D - C) + C$$

Hierbei wird ein Wert x aus einem Intervall $[A, B]$ auf einen Wert x' im Intervall $[C, D]$ abgebildet.

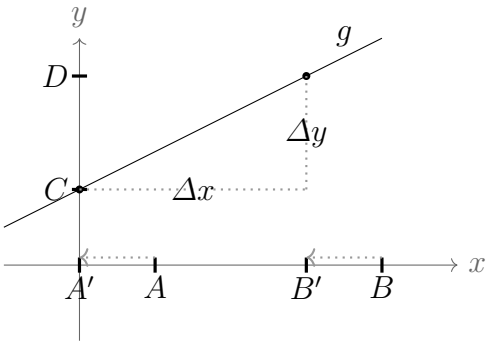


Abbildung 3.3: Abbildung zur Herleitung der Min-Max-Normalisierung.

Wie in Abbildung 3.3 veranschaulicht, erfolgt eine Transformation des Intervalls $[A, B]$ auf der X-Achse in das Intervall $[C, D]$ auf der Y-Achse. Zunächst wird jeder Wert im Intervall $[A, B]$ um den Wert A nach links verschoben. Das bedeutet, dass für eine Zahl x aus dem Intervall $[A, B]$ $x - A$ berechnet wird. Dadurch entsteht das neue Intervall $[A', B']$. Dieser Schritt dient dazu, das Intervall zum Nullpunkt hin zu verschieben.
Anschließend kann die Gerade g leicht definiert werden. Da der Abstand des verschobenen Intervalls zwischen dem minimalen Wert A und dem maximalen Wert B beibehalten wird, ergibt sich die Steigung der Geraden als $\frac{\Delta y}{\Delta x} = \frac{(D-C)}{(B-A)}$.
Der y-Achsenabschnitt der Geraden ist C, da die Gerade das verschobene Intervall

auf das Intervall $[C, D]$ abbilden soll.

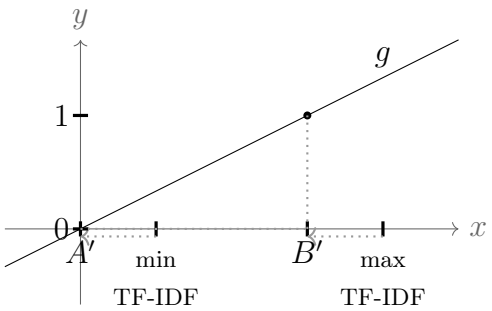


Abbildung 3.4: Abbildung zur Herleitung der Min-Max-Normalisierung der TF-IDF-Werte.

Die TF-IDF-Werte der im Beweis vorkommenden Symbole wurden durch die Min-Max-Normalisierung wie folgt angepasst: Zunächst wurden der minimale und der maximale TF-IDF-Wert für einen Beweis aus der Menge aller berechneten TF-IDF-Werte für einen Beweis ermittelt und dann auf das Intervall $[0, 1]$ abgebildet. Dadurch ergibt sich die folgende Funktion:

$$g(x) = \frac{(x - \text{min TF-IDF})}{(\text{max TF-IDF} - \text{min TF-IDF})}$$

Hierbei bezeichnet „min TF-IDF“ den minimalen TF-IDF-Wert, „max TF-IDF“ den maximalen TF-IDF-Wert aus der Menge aller TF-IDF-Werte eines Beweises, und x bezeichnet den jeweiligen TF-IDF-Wert, der normalisiert werden soll.
Die Funktion wird vereinfacht, indem C und D auf 0 und 1 festgelegt werden. Dadurch wird der y-Achsenabschnitt zu 0, und der Term (C-D) wird zu 1, was aufgrund der Multiplikation weggelassen werden kann.
Für die berechneten Häufigkeitswerte für Symbole, die zwar nicht explizit im Beweis auftauchen, aber dennoch im Beweisprozess verarbeitet wurden, wurde die Min-Max-Normalisierung wie folgt angewendet:
Zunächst wurden die minimale und die maximale Häufigkeit aus der Menge der berechneten Häufigkeiten für einen Beweis ermittelt. Anschließend wurden die Häufigkeitswerte des Beweises auf das Intervall $[-1, 0]$ abgebildet, indem die Gerade, die für die Normalisierung der TF-IDF-Werte verwendet wurde, lediglich an der X-Achse gespiegelt wurde. Dadurch ergibt sich die folgende Funktion:

$$g(x) = \frac{(x - \text{min Häufigkeit})}{(\text{max Häufigkeit} - \text{min Häufigkeit})} \cdot (-1)$$

Hierbei bezeichnet „min Häufigkeit“ die minimale Häufigkeit, und „max Häufigkeit“ die maximale Häufigkeit, aus der Menge aller berechneten Häufigkeitswerte eines Beweises, und x bezeichnet den zu normalisierenden Häufigkeitswert.
Der Grund für die Spiegelung ist, dass die Funktion bei der maximalen Häufigkeit den Wert -1 und bei der minimalen Häufigkeit den Wert 0 haben soll.

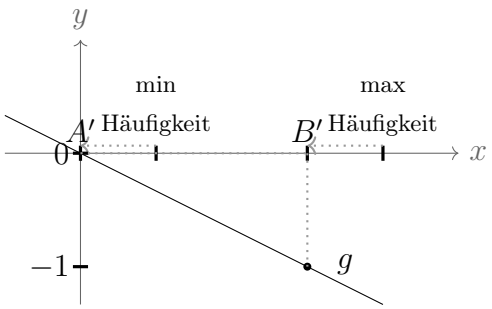


Abbildung 3.5: Abbildung zur Herleitung der Min-Max-Normalisierung für die Häufigkeitswerte.

Die Trainingsdaten befinden sich durch die obige Min-Max-Normalisierung im Intervall $[1, -1]$. Symbole mit einer Bewertung im Bereich $[1, 0]$ sollten ähnlich zum Beweisziel sein, da diese Symbole im Beweis auftreten. Symbole mit einer Bewertung von 0, gelten als neutral. Symbole mit einer Bewertung im Intervall $[-1, 0]$ sollten eine geringe Ähnlichkeit zum Beweisziel aufweisen, da sie vom Beweiser verarbeitet wurden, ohne einen Mehrwert für den Beweis zu liefern.
Es sollte beachtet werden, dass bei jeder Beweisaufgabe sowohl der minimale Häufigkeitswert als auch der minimale TF-IDF-Wert aufgrund der Min-Max-Normalisierung fälschlicherweise mit null bewertet wurde, obwohl diese Werte nicht neutral sind.
Für jede der 2561 Beweisaufgaben wurde beurteilt, wie gut jedes Symbol zu der entsprechenden Aufgabe passt. Somit wurden für jede Beweisaufgabe 2561 Trainingsdaten generiert. Jedoch wurden nur 80 Prozent der insgesamt 7347 Aufgaben, also 5878 Aufgaben, für das Training des Sprachmodells verwendet, während 20 Prozent, das entspricht 1469 Aufgaben, zum Testen des Sprachmodells genutzt wurden. Insgesamt standen somit 15053558 Trainingsdaten für das Training und 3762109 Trainingsdaten für das Testen zur Verfügung.
In dieser Arbeit wurden jedoch nicht alle 3762109 für das Testen vorgesehenen Daten für das Testen des Sprachmodells verwendet.
Um die Vergleichbarkeit zu erhöhen und den Rechenaufwand zu verringern, wurden die Testdaten folgendermaßen generiert: Für jede der 1469 Beweisaufgaben wurden zufällig 10 positiv bewertete Trainingsdaten (Symbole, die gut zu der Beweisaufgabe passen), 15 negativ bewertete Trainingsdaten (Symbole, die nicht gut zu der Beweisaufgabe passen) und 15 neutrale Trainingsdaten ausgewählt. Wenn weniger Trainingsdaten vorhanden waren, wurden entweder alle verfügbaren positiven, alle negativen oder alle neutralen Trainingsdaten verwendet. Dies spiegelt die tatsächlichen Testdaten wieder, die ebenfalls mehr neutrale und negative Symbole als positive Symbole enthalten.
In der folgenden Abbildung sind sowohl die erzeugten Testdaten als auch alle für das Testen vorgesehenen Daten visualisiert.

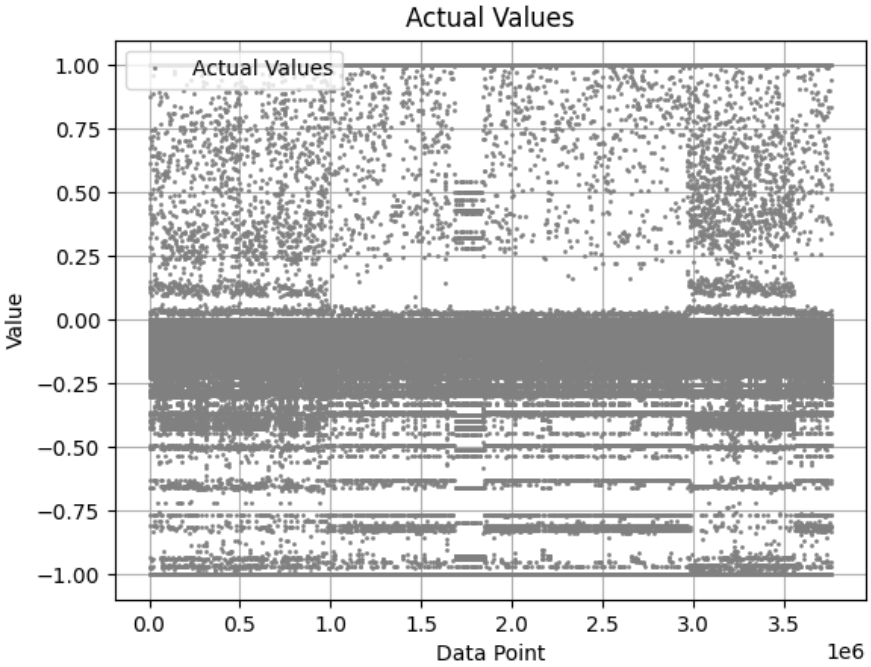


Abbildung 3.6: Alle für das Testen vorgesehenen Daten.

Wie in Abbildung 3.6 zu sehen ist, sind viele der für das Testen vorgesehenen Daten negativ und liegen nahe dem Wert Null.

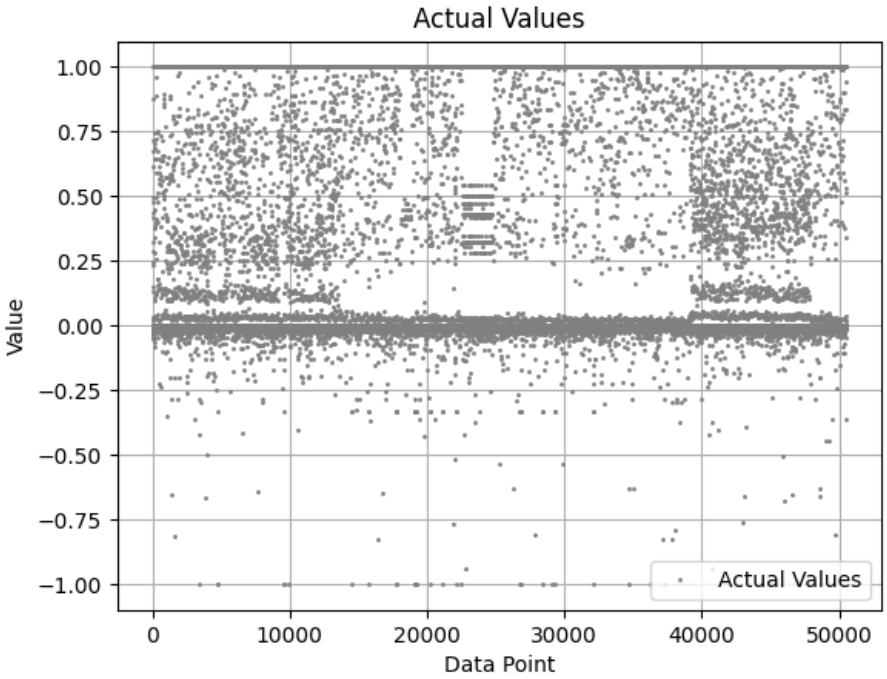


Abbildung 3.7: Für das Testen erzeugte Daten.

Abbildung 3.7 zeigt, dass die viele der generierten Testdaten ebenfalls nahe Null liegen und negativ sind. Daher weisen die generierten Testdaten eine ähnliche Struktur wie die zum Testen vorgesehenen Daten auf.
Diese Trainings- und Testdaten wurden in drei Listen organisiert: Eine für die verbalisierten Ziele, eine für die Symbole und eine für die zugehörigen Werte. Die Organisation erfolgte so, dass das verbalisierte Ziel, das Symbol und der zugehörige Wert jeweils denselben Index in den Listen haben.

3.5 Fine-tuning des Sprachmodells

Der folgende Abschnitt beschreibt das Fine-tuning des Sprachmodells.
Ein Trainingsdatum besteht aus einem Symbolnamen, einem verbalisierten Ziel und einem normalisierten Wert. Das Fine-tuning erfolgt durch die Berechnung der Einbettungsvektoren sowohl für das Ziel als auch den Symbolnamen. Anschließend wird die Kosinusähnlichkeit zwischen diesen Vektoren bestimmt. Der daraus resultierende Verlust, die Abweichung zwischen der Kosinusähnlichkeit und dem normalisierten Wert, wird durch die Anpassung der Gewichte und Biases des Modells reduziert.
Es stellt sich die Frage, inwieweit ein Fine-tuning des ursprünglichen Modells überhaupt notwendig ist. Nachfolgend sind die Vorhersagen des untrainierten Modells für die Testdaten dargestellt.

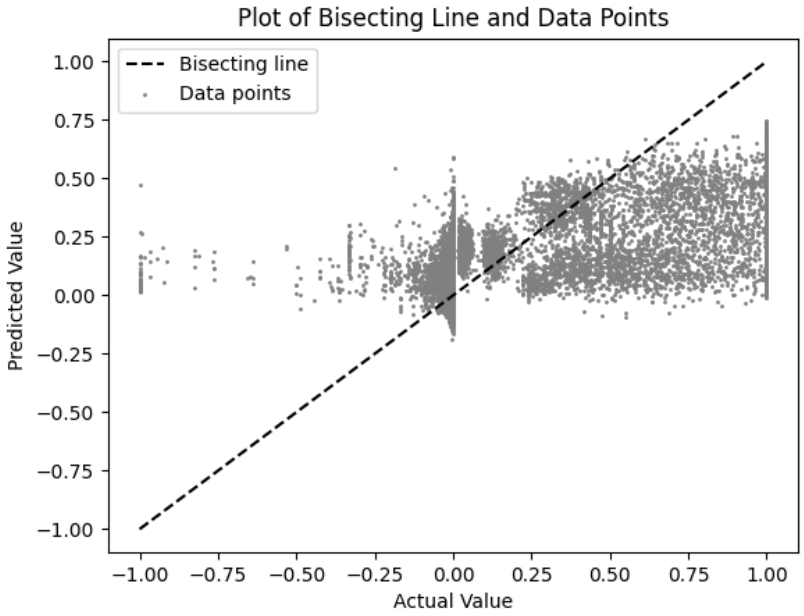


Abbildung 3.8: Vergleich der realen und prognostizierten Werte des untrainierten Sprachmodells.

Wie in Abbildung 3.8 dargestellt, weichen die prognostizierten Werte des untrainierten Sprachmodells deutlich von den tatsächlichen Werten ab. Dies lässt sich daran erkennen, dass nur sehr wenige Datenpunkte in der Nähe der ersten Winkelhalbierenden liegen.
Interessant ist, dass ein Großteil der vom Sprachmodell vorhergesagten Werte (predicted scores) im Bereich zwischen 0 und 0,75 liegt. Dies deutet darauf hin, dass das Modell dazu tendiert, die Symbolnamen überwiegend positiv zu bewerten. Eine plausible Erklärung hierfür wäre, dass das Sprachmodell darauf trainiert wurde, semantische Ähnlichkeiten zwischen Wörtern oder Sätzen in einem Wertebereich von 0 bis 1 zu bewerten.
Auf Basis dieser Daten kann davon ausgegangen werden, dass ein Training des Modells erforderlich ist.
Das Fine-tuning wurde mithilfe von PyTorch durchgeführt. PyTorch ist gemäß der offiziellen Dokumentation [pyt] eine Bibliothek für Deep Learning, die GPUs und CPUs unterstützt.
Zunächst wurden ein PyTorch-Dataset und ein DataLoader gemäß der Anleitung [dat] auf der PyTorch-Webseite erstellt.

```
1  # function to load dataset
2  class CustomDataset(Dataset):
3      def __init__(self, data_file):
4          # loading the data
5          data = torch.load(data_file)
6
7          # creating the attributes for the data
8          self.verbalizedGoals = data['verbalizedGoals']
9          self.symbolNames = data['symbolNames']
10         self.normalizedScores = data['normalizedScores']
11         self.length = len(self.verbalizedGoals)
12
13     # returning the feature and label at a specific index
14     def __getitem__(self, index):
15         verbalizedGoal = self.verbalizedGoals[index]
16         symbolName = self.symbolNames[index]
17         label = self.normalizedScores[index]
18         return verbalizedGoal, symbolName, label
19
20     # returning the sample size
21     def __len__(self):
22         return self.length
```

Listing 3.4: Dataset für die Trainingsdaten.

Die Klasse „`CustomDataset`“ implementiert ein benutzerdefiniertes Dataset für Trainingsdaten. Der Konstruktor (`__init__`) lädt die Trainingsdaten aus einer Datei. Aus diesen Daten werden verbalisierte Ziele, Symbolnamen und normalisierte Werte extrahiert und in entsprechenden Listen innerhalb der Klasse gespeichert. Die Methode „`__getitem__`“ liefert ein 3-Tupel bestehend aus dem verbalisierten Ziel, dem Symbolnamen und dem normalisierten Wert für einen spezifischen Index, der als Argument übergeben wird. Die Methode „`__len__`“ gibt die Anzahl der Elemente im Dataset zurück, indem sie die Länge der Liste der verbalisierten Ziele verwendet. Der `DataLoader` für dieses Dataset wurde folgendermaßen konfiguriert: „`DataLoader(dataset, batch_size=batch_size,`

`shuffle=True)`“. Dabei initialisiert der DataLoader das Dataset mit der spezifizierten `batch_size`, welche die Anzahl der Datensätze pro Batch festlegt, und `shuffle=True`, was angibt, dass die Daten vor jedem Training zufällig gemischt werden.
Die Auswahl der passenden Verlustfunktion ist eine Herausforderung beim Training neuronaler Netze. Es existiert eine Vielzahl von Verlustfunktionen, von denen jede ihre Vor- und Nachteile hat. Abhängig vom Trainingsziel muss die geeignete Verlustfunktion ausgewählt werden, da nicht jede Verlustfunktion zu den gewünschten Ergebnissen führt. Eine Methode, um die richtige Verlustfunktion zu finden, besteht darin, verschiedene auszuprobieren.
Zur Definition einer Verlustfunktion für das Training wurde folgender Ansatz verfolgt, bei dem die Kosinusähnlichkeit zunächst mit der PyTorch-Funktion „`cosineSimilarity`“ berechnet wurde. Anschließend wurde der mittlere quadratische Fehler mittels der Funktion „`mseLoss`“ ermittelt, um die Abweichung zwischen der berechneten Kosinusähnlichkeit und dem normierten Zielwert zu minimieren, wie in der entsprechenden Dokumentation [mse] erläutert.
Die Trainingsdaten zum Testen der Verlustfunktion wurden folgendermaßen erzeugt: Für jede Beweisaufgabe, die für das Training vorgesehen war, wurden jeweils 10 positive, 15 negative und 15 neutral bewertete Symbole zufällig ausgewählt. Waren weniger positive, negative oder neutrale Symbole vorhanden, wurden alle verfügbaren Symbole ausgewählt. Aus diesem Pool an generierten Trainingsdaten wurden anschließend zufällig 5000 Datensätze für das Training ausgewählt.
Die Auswahl der Trainingsdaten reflektiert die Zusammensetzung der eigentlichen Trainingsdaten, in denen ebenfalls mehr neutrale und negative Symbole als positive Symbole vorhanden sind.
Für das Testen des mit der Verlustfunktion trainierten Sprachmodells wurden 5000 zufällig ausgewählte Testdaten aus der Menge der generierten Testdaten verwendet.
Zur detaillierten Untersuchung wurde das Sprachmodell über drei Epochen hinweg trainiert, wobei eine Batchgröße von 8 verwendet wurde und eine Lernrate von 0,00005.
Im Folgenden werden die Trainingsergebnisse vorgestellt, die durch die Anwendung der MseLoss-Funktion erreicht wurden.

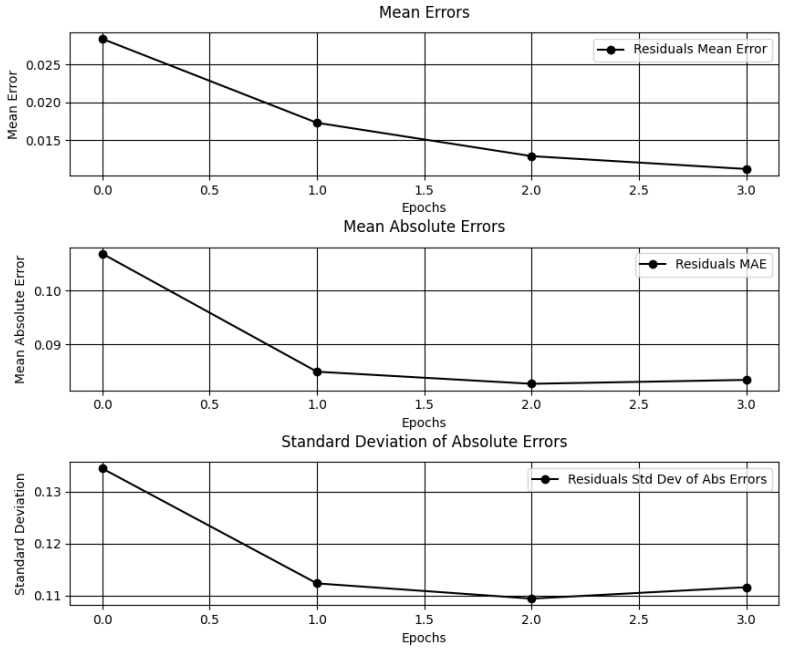


Abbildung 3.9: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen, trainiert mit der MseLoss-Funktion.

Die Abbildung 3.9 zeigt eine deutliche Reduktion sowohl des mittleren absoluten Fehlers als auch der Standardabweichung der absoluten Fehler. Der mittlere Fehler nähert sich von einem positiven Ausgangswert der Null an, was darauf hinweist, dass die Vorhersagen des Sprachmodells die tatsächlichen Werte tendenziell weniger überschätzen. Nach dem Training der zweiten Epoche wurde der geringste mittlere absolute Fehler und eine niedrige Standardabweichung der Fehler beobachtet.
Basierend auf diesen Ergebnissen wurde entschieden, eine detaillierte Auswertung des Modells nach dem Abschluss der zweiten Epoche anhand der Testdaten in der folgenden Abbildung vorzunehmen.

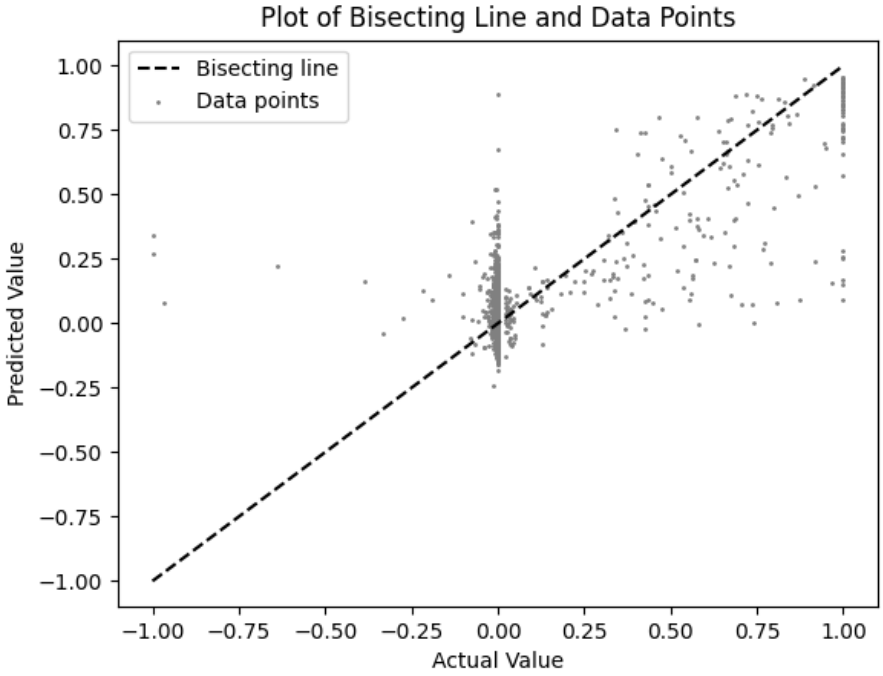


Abbildung 3.10: Vergleich der realen und prognostizierten Werte des durch die MseLoss-Funktion trainierten Sprachmodells.

Die in Abbildung 3.10 dargestellten Daten zeigen eine Annäherung zwischen den vom Modell vorhergesagten Werten und den tatsächlichen Daten, erkennbar an ihrer Nähe zur ersten Winkelhalbierenden.
Es zeigt sich jedoch, dass einige vom Sprachmodell prognostizierte Werte erhebliche Unterschiede zu den tatsächlichen Werten aufweisen, insbesondere bei den tatsächlichen Werten null und eins sowie bei negativen tatsächlichen Werten. Diese signifikanten Unterschiede werden auch als Ausreißer (englisch: Outliers) bezeichnet.
Basierend auf diesen Beobachtungen, wurde die Verwendung der HuberLoss-Funktion als mögliche Alternative in Erwägung gezogen.
Eine detaillierte Beschreibung der Huber Loss-Funktion in PyTorch ist in der Quelle [hub] zu finden. Laut [hub] verwendet die HuberLoss-Funktion einen Schwellenwert, Delta, der zur Fehlerbeurteilung herangezogen wird.
Die Definition der HuberLoss-Funktion in PyTorch lautet:

$$l_n = \begin{cases} 0.5(x_n - y_n)^2, & \text{if } |x_n - y_n| < \text{delta} \\ \text{delta} \cdot |x_n - y_n| - 0.5 \cdot \text{delta}^2, & \text{otherwise} \end{cases}$$

Hierbei ist x_n der vom neuronalen Netz vorhergesagte Wert und y_n der tatsächliche Wert. Über alle berechneten Fehler wird anschließend der Durchschnitt gebildet, um den Gesamtfehler zu erhalten.
Ein wesentlicher Vorteil dieser Funktion ist ihre geringere Empfindlichkeit gegenüber Ausreißern in den Trainingsdaten.

Nachfolgend sind die Trainingsergebnisse dargestellt, die mithilfe der HuberLoss-Funktion und einem Delta-Wert von 0,3 erreicht wurden. Der gewählte Wert von 0,3 basiert auf der Beobachtung in Abbildung 3.10, dass viele Datenpunkte einen absoluten Fehler von weniger als 0,3 aufweisen, bzw. einen Abstand von 0,3 zur Winkelhalbierenden haben. Für einen Abstand größer als 0,3 scheint die Anzahl der Datenpunkte deutlich abzunehmen, was auf das Vorhandensein von Ausreißern hindeutet.

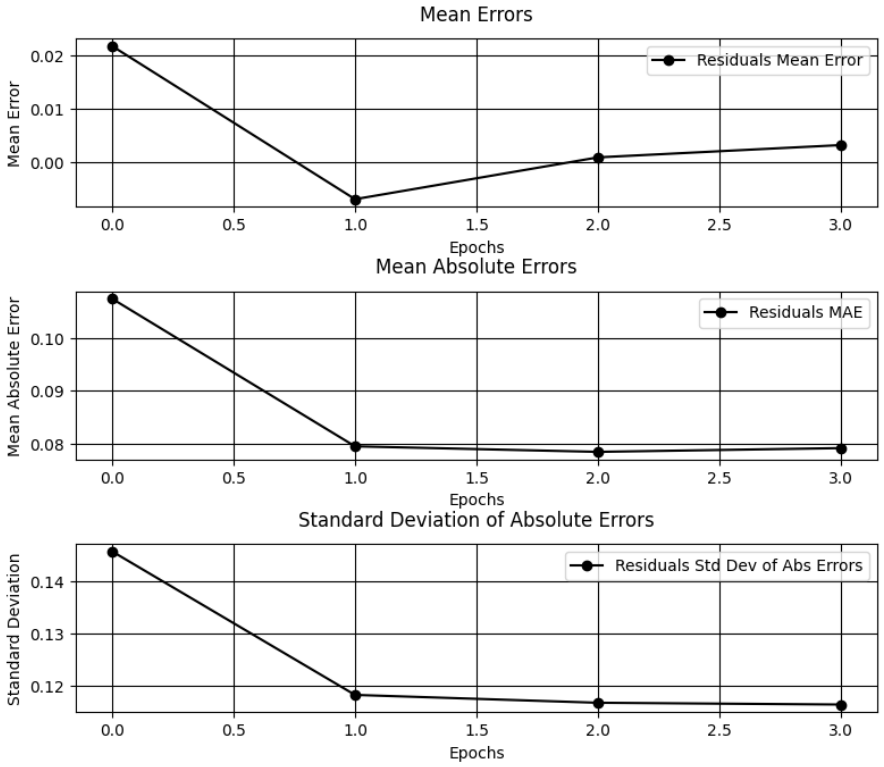


Abbildung 3.11: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen, trainiert mit der HuberLoss-Funktion.

Wie in Abbildung 3.11 dargestellt, weisen sowohl der mittlere absolute Fehler als auch die Standardabweichung der absoluten Fehler nur geringfügige Veränderungen im Vergleich zum Training mit der MseLoss-Funktion auf. Der mittlere Fehler ist nicht vollständig stabil, was darauf hindeutet, dass die Modellvorhersagen gelegentlich zu niedrig oder zu hoch sind. Das trainierte Sprachmodell nach der zweiten Epoche zeigt den niedrigsten mittleren Fehler und eine geringe Standardabweichung der absoluten Fehler.
Aus diesem Grund wird die Auswertung des Sprachmodells nach dieser Epoche mit den Testdaten im Folgenden dargestellt.

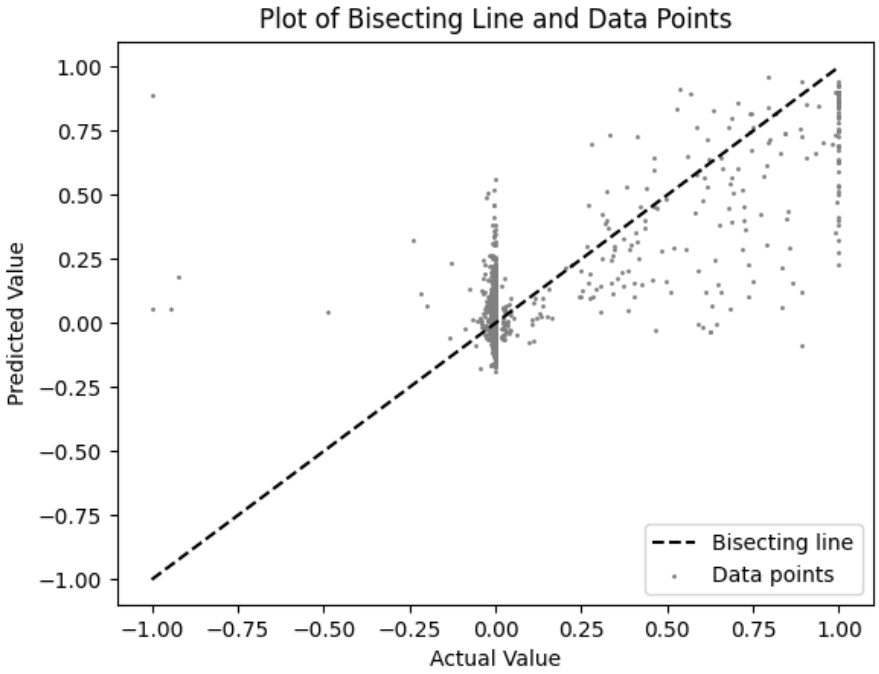


Abbildung 3.12: Vergleich der realen und prognostizierten Werte des durch die HuberLoss-Funktion trainierten Sprachmodells.

Wie in Abbildung 3.12 dargestellt, sind die Verbesserungen hinsichtlich der Vorhersagequalität marginal. Dies wird dadurch ersichtlich, dass kaum mehr Datenpunkte in unmittelbarer Nähe der ersten Winkelhalbierenden liegen, was eine Annäherung der Vorhersagen an die tatsächlichen Werte andeuten würde.
Diese Beobachtung könnte auf den gewählten Delta-Wert der HuberLoss-Funktion zurückgehen. Eine Anpassung dieses Wertes könnte potenziell zu besseren Ergebnissen führen.
Aufgrund der Erkenntnisse über die verschiedenen Loss-Funktionen wurde beschlossen, das Training des Sprachmodells mit der MSELoss-Funktion durchzuführen.
Zu Beginn des Trainings wurden das Modell und der Tokenizer mittels der Hugging Face-Bibliothek geladen. Dies geschah durch die Verwendung der Methoden „`AutoModel.from_pretrained(pretrained_model_name_or_path)`“ und „`AutoTokenizer.from_pretrained(pretrained_model_name_or_path)`“.
Der Parameter „`pretrained_model_name_or_path`“ kann auf der Hugging Face-Seite des entsprechenden Modells gefunden werden. Beispielsweise lautet der Pfad für das in dieser Arbeit verwendete Modell MiniLM-L6-v2:
`''sentence-transformers/all-MiniLM-L6-v2''` (siehe [all]).
Weitere Details zu diesen Methoden sind in der Hugging Face-Dokumentation unter [loa] zu finden.
Für das Batch-Training wurde der AdamW-Optimizer konfiguriert, um die Effizienz des Trainings zu steigern.
Die Konfiguration erfolgte mittels „`AdamW(model.parameters(), lr=5e-5)`“,

wobei „`model.parameters()`“ die vom Optimierer anzupassenden Modellparameter, typischerweise Gewichte und Bias-Werte der Netzwerkschichten, bezeichnet. Die festgelegte Lernrate „`lr=5e-5`“ reguliert das Ausmaß der Parameteranpassungen während des Trainings. Detaillierte Informationen zum AdamW-Optimizer sind auf der PyTorch-Website [ada] zu finden.
Basierend auf der Anleitung zur Erstellung eines Training-Loops auf der PyTorch-Webseite gemäß [tra] wurde der folgende Training-Loop für das Sprachmodell entwickelt.

```
# specifying the criterion
criterion = MSELoss()

# set the model into training mode
model.train()

# training loop
for epoch in range(epochs):

    for batch in dataloader
        # extracting the goals, symbols and labels
        verbalizedGoals, symbolNames, labels = batch

        # zeroing the gradients
        optimizer.zero_grad()

        # tokenize the goals and the symbols
        goals = [goal for goal in verbalizedGoals]
        tokenizedGoals = tokenizer(goals, padding=True,
                                   truncation=True, return_tensors='pt')
        symbols = [symbol for symbol in symbolNames]
        tokenizedSymbols = tokenizer(symbols, padding=True,
                                     truncation=True, return_tensors='pt')

        # generate a tensor for the labels
        labels = [label.float() for label in labels]
        labels = torch.tensor(labels, device=device)

        # forward path of the goals and symbols
        goalEmbeddings = model(**tokenizedGoals)
        symbolEmbeddings = model(**tokenizedSymbols)

        # calculating the cosine similarity between the goals and symbols
        cos = CosineSimilarity(dim=1)
        cosSim = cos(goalEmbeddings, symbolEmbeddings)

        # calculating the loss
        loss = criterion(cosSim, labels)

        # do backpropagation
        loss.backward()

        # optimize the weights
        optimizer.step()
```

Listing 3.5: Vereinfachte Darstellung des Trainings-Loops.

Der Training-Loop wurde vereinfacht, indem unwichtige Details wie das Speichern des Trainingsstandes nach jeder Epoche, das Mitteln der Einbettungsvektoren, das Speichern des Fehlers in einer Variablen und das Verschieben der Tensoren auf eine Grafikkarte aus Platzgründen weggelassen wurden, da sie für die Logik nicht relevant sind.

Abschließend wurde das trainierte Modell gemäß [mod] gespeichert, in Form eines „state_dict“, welches die erlernten Parameter enthält.
Das Sprachmodell wurde zunächst mit allen 15053558 Trainingsdaten trainiert. Das Training umfasste drei Epochen mit einer Batch-Größe von 44 und einer Lernrate von 0,00005.
Im Folgenden ist das Ergebnis des Trainings dargestellt.

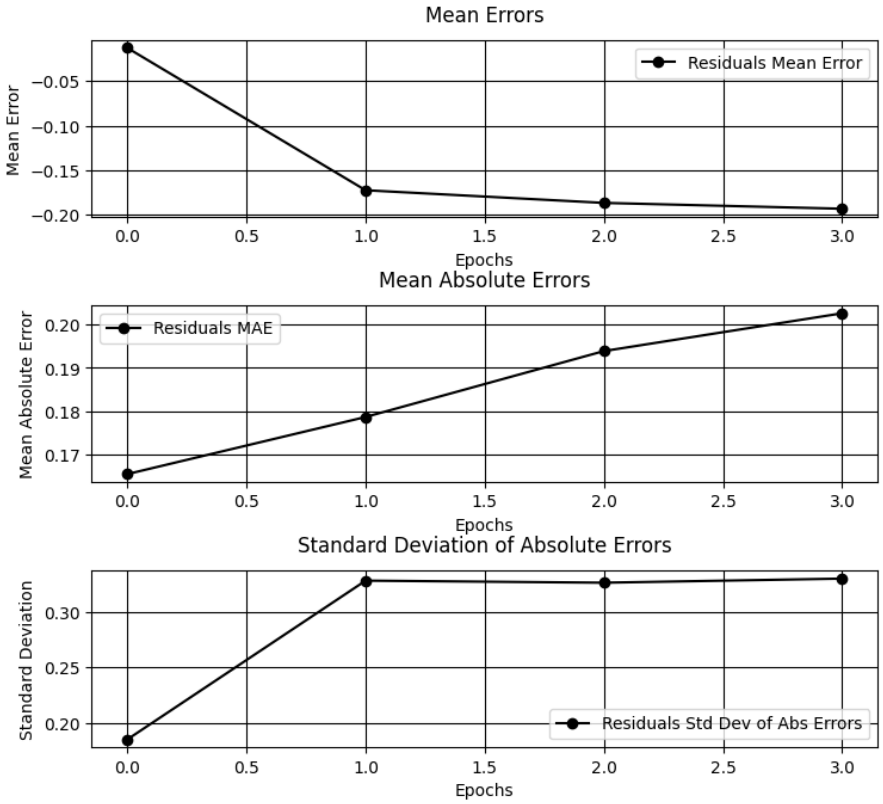


Abbildung 3.13: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen des mit allen Trainingsdaten trainierten Sprachmodells.

Wie in Abbildung 3.13 ersichtlich, steigen der mittlere absolute Fehler sowie die Standardabweichung der absoluten Fehler kontinuierlich an, was darauf hindeutet, dass die Leistung des trainierten Netzwerks abnimmt. Zudem wird der mittlere Fehler stetig negativer, was darauf hinweist, dass das Netzwerk dazu tendiert, die tatsächlichen Werte zunehmend zu unterschätzen.
Nach der ersten trainierten Epoche wies das Sprachmodell den geringsten mittleren absoluten Fehler auf.
Aus diesem Grund wird die Auswertung des Sprachmodells nach dieser Epoche mit den Testdaten im Folgenden dargestellt.

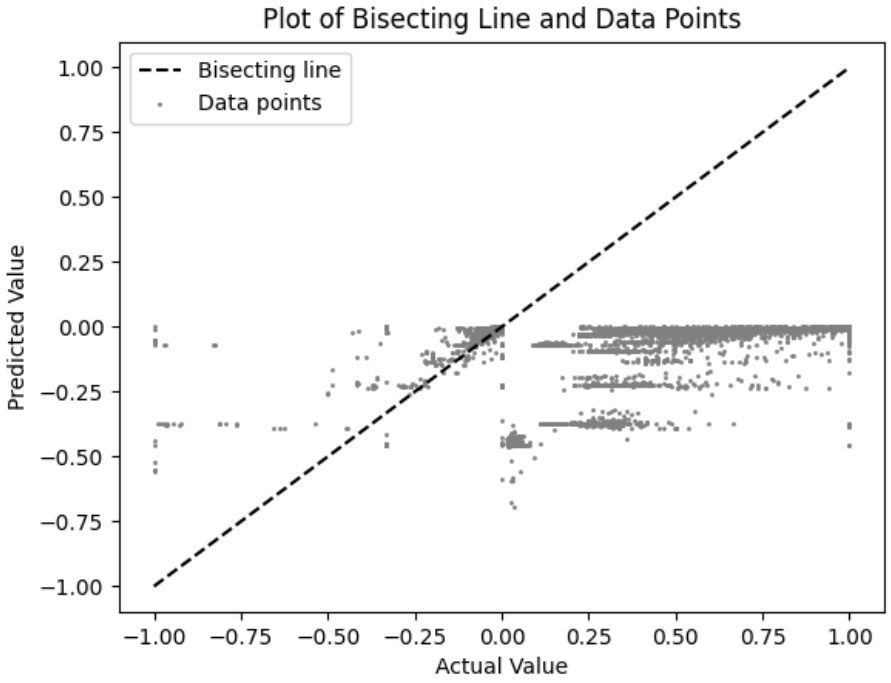


Abbildung 3.14: Vergleich der realen und prognostizierten Werte des mit allen Trainingsdaten trainierten Sprachmodells.

In der Abbildung 3.14 ist ersichtlich, dass die Qualität der Vorhersagen signifikant abgenommen hat. Nahezu alle Vorhersagen liegen im Bereich von -0,4 bis 0.
Eine mögliche Erklärung hierfür könnte sein, dass die Trainingsdaten sehr unausgeglichen waren.
Dies ist ein Problem was normalerweise mehr aus dem Bereich der Klassifzierung bekannt ist als aus den Bereich der Regression.
Der Trainingsdatensatz umfasst insgesamt 15053558 Symbole mit zugehörigem Beweisziel, von denen 40449 (0,27%) positiv, 13769266 (91,47%) negativ und 1243843 (8,26%) neutral sind. Dies zeigt eine deutliche Dominanz von negativen Beispielen gegenüber den positiven und neutralen Symbolen. Der Grund dafür ist, dass für die positiven Trainingsdaten ausschließlich die Symbole verwendet wurden, die im Beweis vorkommen, was meist eine sehr begrenzte Anzahl ist. Im Gegensatz dazu wurden für die negativen Trainingsdaten alle Symbole herangezogen, die vom Beweiser verarbeitet, jedoch nicht im Beweis enthalten sind, was in der Regel deutlich mehr Symbole umfasst. Alle anderen Symbole wurden als neutral eingestuft und erhielten den Wert null. Da der Beweiser viele Klauseln pro Beweis verarbeitete und nahezu jedes Symbol in mindestens einer dieser Klauseln vorkam, ist die Anzahl der neutral bewerteten Symbole relativ gering.
Viele Trainingsdaten haben auch einen Wert nahe null, da es nur wenige Symbolnamen gibt, die perfekt zum Beweisziel passen und die Mehrheit der Symbolnamen weder sehr gut noch sehr schlecht zu dem Beweisziel passen.
Aufgrund dieser Gegebenheiten tendiert das Sprachmodell möglicherweise dazu, die Ähnlichkeit eines Symbols mit einem zu beweisenden Ziel negativ und nahe

null zu bewerten.
In [TBRP15] wird das Problem näher erläutert. Gemäß [TBRP15] (S. 466) ist das Problem wie folgt definiert: Gegeben ist eine Menge von N Eingabe-Ausgabe-Beispiel-Paaren $(x_1, y_1), \ldots, (x_N, y_N)$. Das Ziel der Regression besteht darin, eine Funktion $Y = f(x)$ anzunähern. Allerdings besteht das eigentliche Ziel darin, eine Funktion $Y' = f(x)$ zu finden, wobei die Menge Y' eine Teilmenge von Y ist mit extrem seltenen Werten aus Y.
Dies ist relevant, da die Auswahl der Symbole, die am besten zu einem spezifischen Ziel passen, sowie jene, die am wenigsten geeignet sind, sehr seltene Werte darstellen. Die Mehrheit der Symbole liefert nur begrenzt aussagekräftige Informationen über ihre Eignung.
In der folgenden Abbildung werden die normalisierten tatsächlichen Werte aller Trainingsdaten dargestellt. Aufgrund der Herausforderung, eine so große Datenmenge übersichtlich zu präsentieren, wurden jeweils fünf Datenpunkte zu einer Gruppe zusammengefasst und der Durchschnitt dieser Gruppe gebildet.

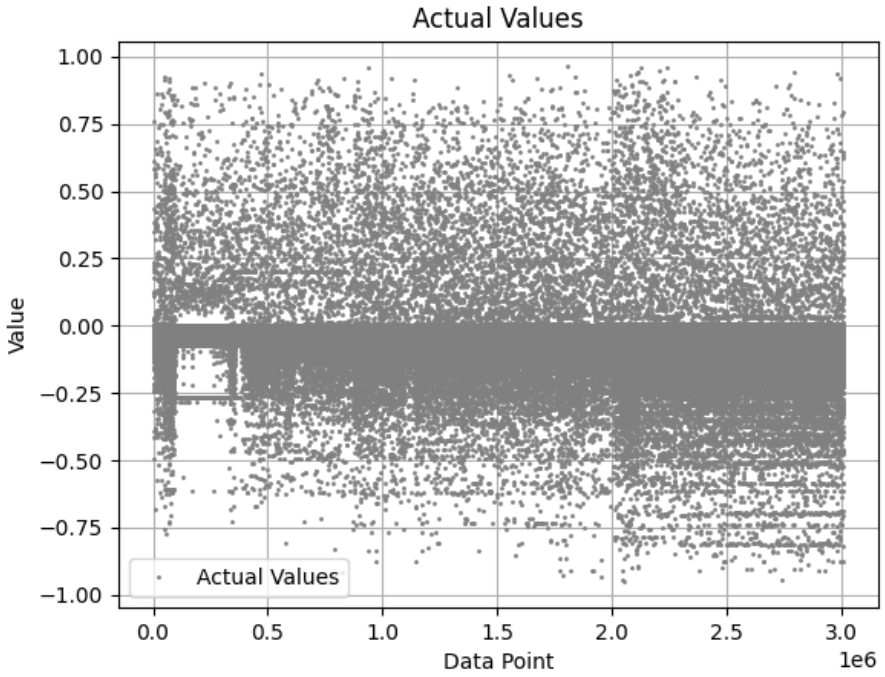


Abbildung 3.15: Visualisierung des unausgeglichenen Trainingsdatensatzes, der alle für das Training vorgesehenen Daten beinhaltet.

Wie in Abbildung 3.15 dargestellt, sind die meisten Trainingsdaten im Bereich von 0 bis -0,4 konzentriert, welcher überwiegend auch vom trainierten Netzwerk prognostiziert wurde.
Dies könnte darauf hindeuten, dass das Sprachmodell nicht in der Lage ist, die volle Komplexität der Daten abzubilden, was auf Underfitting schließen lässt.
Eine denkbare Lösung wäre der Einsatz eines alternativen Modells mit einer größeren Anzahl an Parametern, um die Datenkomplexität besser zu erfassen.

Jedoch ist die Existenz eines solchen Modells unsicher, und der erhöhte Trainingsaufwand würde zudem mehr Zeit in Anspruch nehmen. Zudem birgt ein komplexeres Modell das Risiko des Overfittings, was die Generalisierbarkeit auf neue Daten verschlechtern könnte.
Eine weitere vorgeschlagene Methode, das sogenannte „Under-sampling“, welches in [TBRP15] (S. 469) diskutiert wird, beinhaltet das Entfernen der häufigsten Werte aus den Trainingsdaten. Das Risiko hierbei ist allerdings, dass potenziell relevante Daten verloren gehen, wodurch die Trainingsdaten die Realität möglicherweise nicht mehr adäquat repräsentieren. Es ist daher wichtig, sorgfältig abzuwägen, wie viele Daten entfernt werden sollten, um eine ausgewogene Trainingsgrundlage zu erhalten.
Auf der Grundlage dieser Überlegungen wurden die Trainingsdaten wie folgt angepasst:
Für jede Beweisaufgabe, die für das Training vorgesehen ist, wurde die Anzahl der positiven Trainingsdaten bestimmt und mit n bezeichnet. Ein Faktor k wurde eingeführt, um das Verhältnis von negativen und neutralen zu positiven Daten zu definieren. Dieser Faktor legt fest, dass die Anzahl der negativen und neutralen Trainingsbeispiele $n \cdot k$ beträgt. Aus den negativen und neutralen Trainingsdaten wurden $n \cdot k$ Daten zufällig ausgewählt, um verwendet zu werden. Sollte $n \cdot k$ die verfügbaren negativen und neutralen Trainingsdaten übersteigen, so wurden alle verfügbaren negativen und neutralen Daten verwendet. Daher bestanden die Trainingsdatensätze aus n positiven und $n \cdot k$ negativen und neutralen Beispielen.
Es stellt sich nun die Frage, welcher Wert für k optimal ist und welchen Einfluss dieser Faktor auf das Training hat. Aus diesem Grund wurden in dieser Arbeit Trainingsdatensätze mit Verhältnissen von $k = 2$ und $k = 5$ untersucht. Dabei entspricht $k = 2$ einem Verhältnis, bei dem die Anzahl der negativen und neutralen Trainingsdaten doppelt so hoch ist wie die der positiven. Bei $k = 5$ ist die Anzahl der negativen und neutralen Trainingsdaten fünffach so hoch wie die der positiven.
Diese Datensätze sollten deutlich ausgewogener sein, da sie einen deutlich geringeren Anteil an negativen und neutralen Trainingsdaten aufweisen. Dennoch dürften sie die Charakteristiken des ursprünglichen Datensatzes weiterhin angemessen widerspiegeln. Dies wird durch die zufällige Auswahl der negativen und neutralen Trainingsdaten erreicht, was zur Folge hat, dass immer noch viele Daten nahe dem Nullpunkt liegen sollten.
Die folgenden Abbildungen visualisieren den Datensatz für die Parameter $k = 2$ und $k = 5$.
Ein weiterer Ansatz, der eventuell vielversprechend ist, besteht darin, den Trainingsdaten basierend auf ihrem Wert zufällige Störungen hinzuzufügen. Konkret könnte man bei Daten, die den Wert null haben, einen kleinen zufälligen positiven oder negativen Wert addieren. Bei Daten mit dem Wert eins könnte man einen kleinen zufälligen Wert subtrahieren und bei solchen mit dem Wert minus eins einen kleinen zufälligen Wert hinzufügen. Diese Methode, könnte helfen, Overfitting zu vermeiden. Overfitting tritt auf, wenn ein Modell die Trainingsdaten zu genau lernt, einschließlich des Rauschens und der Ausreißer, was zu einer schlech-

teren Generalisierung auf neuen, unbekannten Daten führt. Durch die Einführung von Störungen werden die Trainingsdaten weniger identisch, und das Modell wird gezwungen, sich auf die allgemeineren Muster zu konzentrieren, statt sich auf spezifische Datenpunkte zu überanpassen.
Dieser Ansatz wurde in dieser Arbeit jedoch nicht weiter untersucht, und es wurde keine Literatur im Bezug auf Regressionsprobleme gefunden.

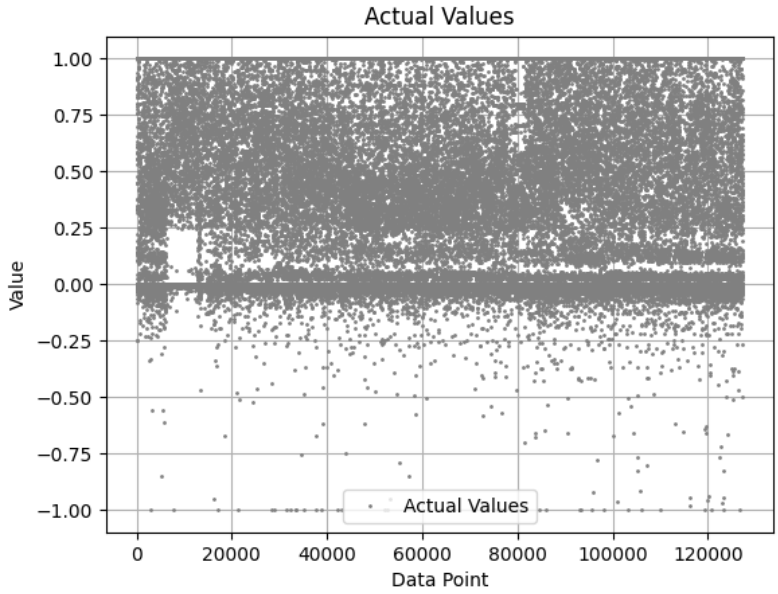


Abbildung 3.16: Visualisierung des durch Undersampling erzeugten Trainingsdatensatzes mit dem Parameter von $k = 2$.

Die Abbildung 3.16 scheint nicht der Realität zu entsprechen, da der Eindruck erweckt wird, es gäbe mehr positive als negative Beispiele im Datensatz. Tatsächlich jedoch, wie bereits erwähnt, enthält der Datensatz doppelt so viele neutrale und negative Beispiele wie positive. Diese Wahrnehmung resultiert daraus, dass die neutralen und negativen Datenwerte größtenteils um den Nullpunkt konzentriert sind, was in der Abbildung zu einer starken Überlagerung führt. Dies ist ähnlich wie dem ursprüngliche Trainingsdatensatz in Abbildung 3.15. Dies deutet darauf hin, dass der durch Downsampling modifizierte Datensatz immer noch Charakteristika des Originaldatensatzes widerspiegelt. Trotzdem weist der angepasste Trainingsdatensatz eine ausgewogenere Verteilung auf, mit vielen Datenpunkten in der Nähe von Null.

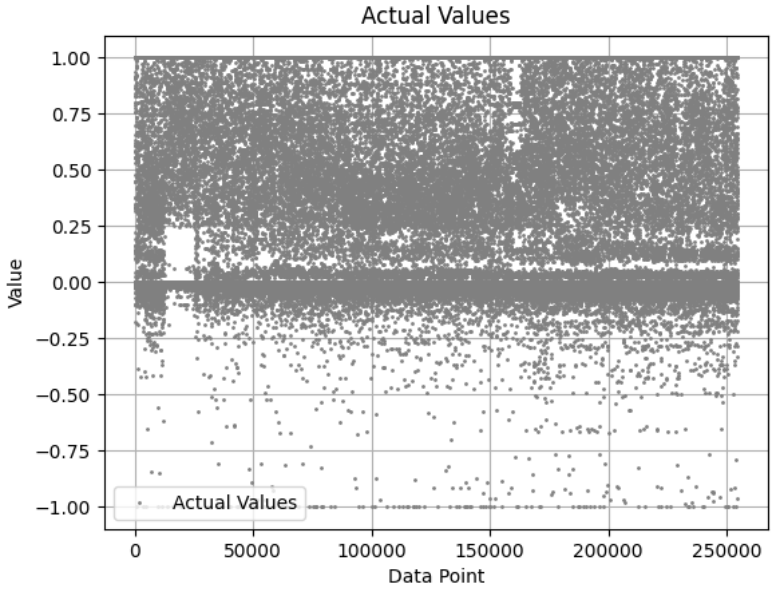


Abbildung 3.17: Visualisierung des durch Undersampling erzeugten Trainingsdatensatzes mit dem Parameter von $k = 5$.

Die Abbildung 3.17 scheint ebenfalls nicht der Realität zu entsprechen, da der Eindruck erweckt wird, es gäbe mehr positive als negative Beispiele im Datensatz. Tatsächlich jedoch, wie bereits erwähnt, enthält der Datensatz fünfmal so viele neutrale und negative Beispiele wie positive.
Diese Wahrnehmung resultiert ebenfalls daraus, dass die neutralen und negativen Datenwerte größtenteils um den Nullpunkt konzentriert sind, was in der Abbildung zu einer starken Überlagerung führt.
Abbildung 3.17 verdeutlicht, dass eine Erhöhung des Parameters k erwartungsgemäß zu einer Zunahme negativer und neutraler Trainingsdaten führt. Es stellt sich die Frage, ob eine Erhöhung des Parameters k und die daraus resultierende Zunahme negativer und neutraler Daten beim Fine-tuning dazu beitragen könnte, dass das Sprachmodell präzisere Vorhersagen trifft.
Um dies zu beantworten, wurde das Sprachmodell mit beiden Trainingsdatensätzen trainiert. Nachfolgend werden die Ergebnisse dieses Trainings präsentiert. Das Modell wurde über fünf Epochen mit einer Batch-Größe von acht und einer Lernrate von 0,00005 trainiert.
Im Folgenden sind die Ergebnisse des Trainings dargestellt.

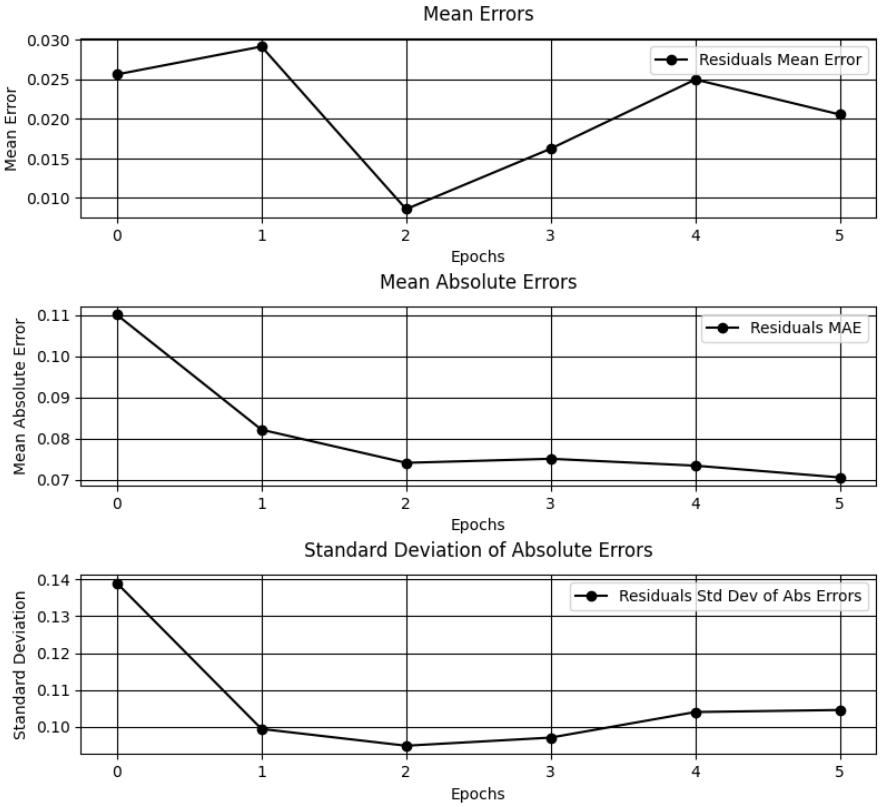


Abbildung 3.18: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen des trainierten Sprachmodells mit dem durch Undersampling mit den Parameter $k = 2$ erstellten Trainingsdatensatzes.

Wie in Abbildung 3.18 dargestellt, zeigt sich eine Reduktion sowohl des mittleren absoluten Fehlers als auch der Varianz der absoluten Fehler. Der mittlere Fehler weist Schwankungen auf, bleibt jedoch durchweg positiv, was darauf hinweist, dass das Sprachmodell dazu neigt, die tatsächlichen Daten zu überschätzen.
Das nach der fünften Epoche trainierte Sprachmodell erweist sich als besonders effektiv, da es den geringsten mittleren absoluten Fehler sowie eine niedrige Varianz dieser Fehler aufweist.
Aus diesem Grund wird die Auswertung des Sprachmodells nach dieser Epoche mit den Testdaten im Folgenden dargestellt.

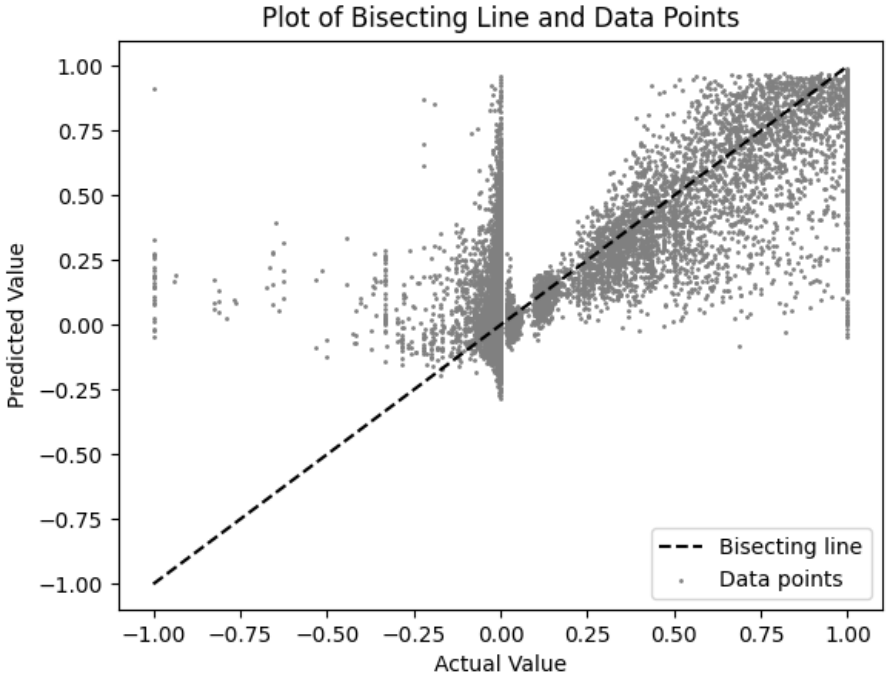


Abbildung 3.19: Vergleich der realen und prognostizierten Werte des mit trainierten Sprachmodells mit dem durch Undersampling mit den Parameter $k = 2$ erstellten Trainingsdatensatzes.

Wie in Abbildung 3.19 ersichtlich, liefert das Modell bereits recht präzise Vorhersagen, was daran erkennbar ist, dass viele Datenpunkte auf oder nahe der Winkelhalbierenden liegen.
Ein erkennbares Problem besteht darin, dass der tatsächliche Wert 0 deutlich überschätzt wird, ebenso wie die negativen tatsächlichen Werte, während der tatsächliche Wert 1 erheblich unterschätzt wird.
Das bedeutet, dass die Vorhersagen der tatsächlichen Werte, wenn diese Null, Eins oder negativ sind, ungenauer ausfallen.
Dies könnte daran liegen, dass durch das Undersampling viele negative und neutrale Trainingsdaten entfernt wurden. Eine Erhöhung des Parameters k könnte potenziell zu genaueren Vorhersagen führen.

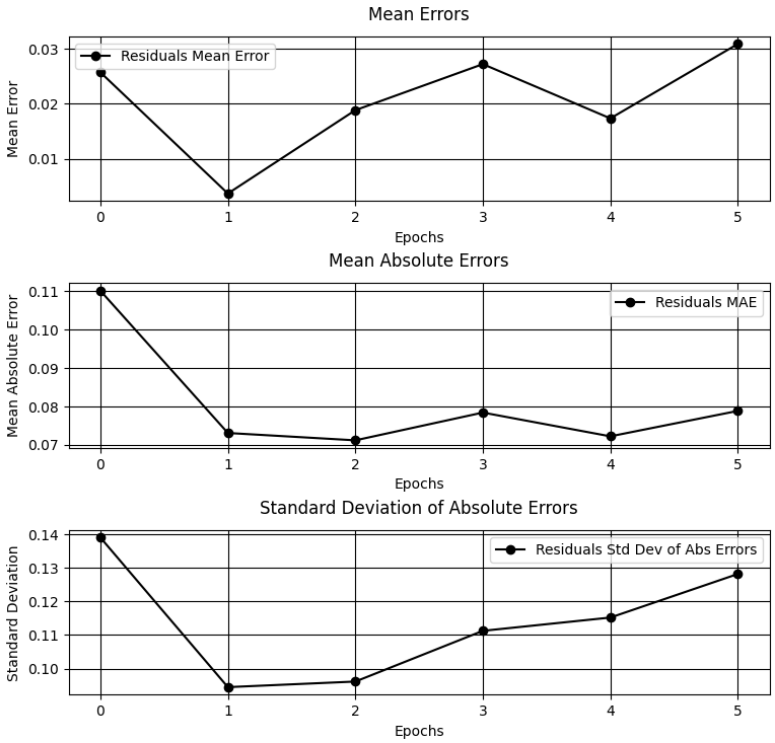


Abbildung 3.20: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen des trainierten Sprachmodells mit dem durch Undersampling mit den Parameter $k = 5$ erstellten Trainingsdatensatzes.

Die in Abbildung 3.20 dargestellten Daten zeigen ebenfalls eine deutliche Reduktion des mittleren absoluten Fehlers. Obwohl die Varianz der mittleren Fehler signifikante Schwankungen aufweist, bleiben diese im niedrigen Bereich. Es ist festzustellen, dass der mittlere Fehler konstant positiv ist, was darauf hindeutet, dass das Modell die tatsächlichen Werte ebenfalls tendenziell überschätzt.
Nach Abschluss der zweiten Epoche zeigt das trainierte Sprachmodell den niedrigsten mittleren absoluten Fehler und eine niedrige Varianz auf.
Aus diesem Grund wird die Auswertung des Sprachmodells nach dieser Epoche mit den Testdaten im Folgenden dargestellt.

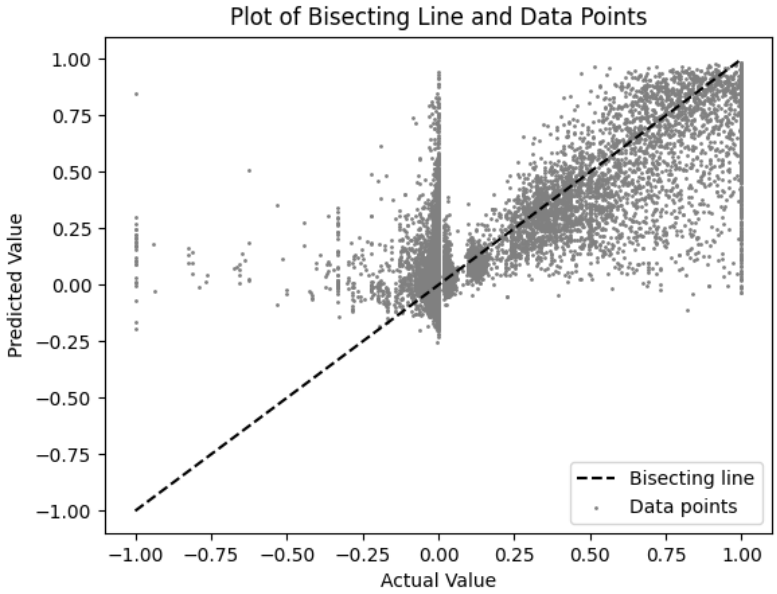


Abbildung 3.21: Vergleich der realen und prognostizierten Werte des mit trainierten Sprachmodells mit dem durch Undersampling mit den Parameter $k = 5$ erstellten Trainingsdatensatzes.

Wie in Abbildung 3.21 ersichtlich, sind die Vorhersagen ziemlich präzise, erkennbar daran, dass viele Datenpunkte auf oder nahe der Winkelhalbierenden liegen.
Bei den Vorhersagen des Sprachmodells tritt das gleiche bereits oben beschriebene Problem auf: Der tatsächliche Wert 0 wird deutlich überschätzt, ebenso wie die negativen tatsächlichen Werte, während der tatsächliche Wert 1 erheblich unterschätzt wird.
Das bedeutet ebenfalls, dass die Vorhersagen der tatsächlichen Werte, wenn diese Null, Eins oder negativ sind, ungenau ausfallen.
Dies könnte ebenfalls daran liegen, dass durch das Undersampling viele negative und neutrale Trainingsdaten entfernt wurden. Eine Erhöhung des Parameters k könnte potenziell zu genaueren Vorhersagen führen.
Um einen besseren Vergleich der verschiedenen Trainingsansätze zu ermöglichen, wurden die Abbildungen 3.18 und 3.20, welche die trainierten Epochen zeigen, nebeneinander dargestellt.

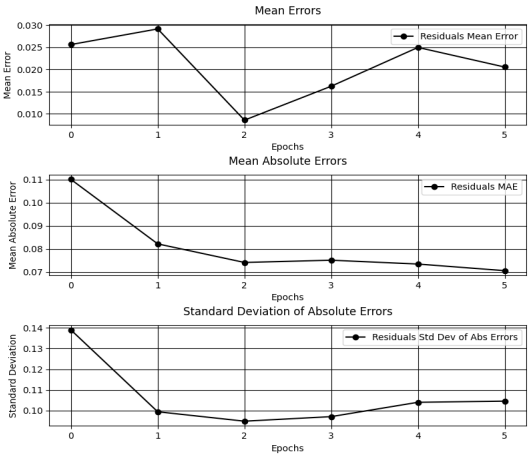


Abbildung 3.22: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen des trainierten Sprachmodells mit dem durch Undersampling mit den Parameter k = 2 erstellten Trainingsdatensatzes.

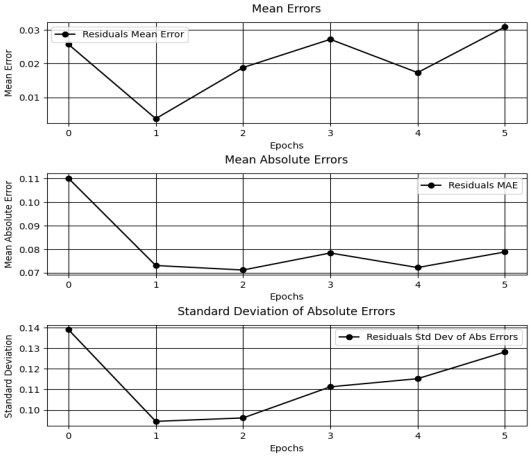


Abbildung 3.23: Darstellung des mittleren Fehlers, des absoluten Fehlers und der Standardabweichung der absoluten Fehler über die Epochen des trainierten Sprachmodells mit dem durch Undersampling mit den Parameter k = 5 erstellten Trainingsdatensatzes.

Ein Vergleich zwischen Epoche vier aus Abbildung 3.22 und Epoche zwei aus Abbildung 3.23 zeigt, dass der mittlere absolute Fehler in 3.23 nach der zweiten Epoche etwas niedriger ist als in Abbildung 3.22 nach der vierten Epoche. Zudem ist die Varianz der absoluten Fehler in 3.23 nach der zweiten Epoche deutlich geringer als in 3.22 nach der vierten Epoche.
Daher scheint die Erhöhung des Parameters auf K=5 positive Auswirkungen gehabt zu haben, da das durch Undersampling mit dem Parameter k=5 trainierte Modell für die Testdaten verbesserte Ergebnisse zeigt als das mit dem Parameter k=2 trainierte Modell.
Das Sprachmodell, das mit einem Datensatz trainiert wurde, welcher mittels Undersampling und dem Parameter k=5 erzeugt wurde, zeigte nach der zweiten Epoche auf Basis der obigen Daten die besten Ergebnisse. Deshalb wurde das Modell nach dieser Epoche für den weiteren Einsatz verwendet.

3.6 Evaluation

In diesem Kapitel wird ein Vergleich durchgeführt zwischen dem Standardbeweisverfahren des Eprovers, das automatisch verschiedene Heuristiken mittels des Parameters `-auto` nutzt, und einer eigens entwickelten Heuristik.
Diese entwickelte Heuristik ermittelt die besten n und die schlechtesten k Symbole basierend auf einem Ähnlichkeitswert, der von einem Sprachmodell in Bezug auf eine Beweisaufgabe berechnet wird.
Die ausgewählten Symbole und ihre Ähnlichkeitswerte werden anschließend dem

Eprover übergeben, um den Beweisprozess zu steuern.
Die Entwicklung der Heuristik basiert auf dem oben beschriebenen Sprachmodell, das mit einem durch Undersampling und den Parameter k=5 erzeugten Datensatz trainiert wurde.
Abschließend werden die Ergebnisse des Vergleichs vorgestellt und diskutiert sowie die Rahmenbedingungen präsentiert.

3.6.1 Durchführung des Vergleichs

Die Durchführung des Vergleichs erfolgte wie folgt.
In den vorhergehenden Abschnitten dieser Arbeit wurde das in [Sch22] vorgestellte Mapping zur Übersetzung der Symbole von Adimen-SUMO in natürliche Sprache bereits erwähnt. Dieses Mapping stellt eine Zuordnung von 3179 logischen Symbolen zu 2561 Wörtern und Sätzen in natürlicher Sprache her.
Zunächst wurde für jede der 2561 Übersetzungen genau ein Embedding-Vektor mit Hilfe des Sprachmodells berechnet. Anschließend wurde jedes Embedding den entsprechenden logischen Symbolen zugeordnet, die mit der jeweiligen Übersetzung übersetzt wurden.
Die vorberechneten Embeddings mit den jeweiligen logischen Symbolen wurden schließlich in einer Datei gespeichert.

Beispiel 3.10. Angenommen, p__citizen ist ein logisches Symbol aus Adimen-SUMO, welches mit *citizen* übersetzt wird. Zunächst wird der Einbettungsvektor

$$\vec{v} = \begin{bmatrix} 7 \\ 2 \\ 15 \end{bmatrix}$$

für *citizen* durch das Sprachmodell ermittelt. Dieser Vektor wird dann dem entsprechenden logischen Symbol p__citizen zugeordnet, das übersetzt wurde.

Ein Programm wurde entwickelt, das das Embedding eines verbalisierten Beweisziels entgegennimmt und die n am besten passenden sowie die k am schlechtesten passenden Symbole mit deren Ähnlichkeitswerten zurückgibt.
Dies erfolgt durch die Berechnung der Kosinusähnlichkeit zwischen jedem der vorberechneten Embeddings für die Symbole und dem Embedding des verbalisierten Ziels.
Ursprünglich war geplant, dies mit der Bibliothek Gensim zu implementieren. Gensim ist eine Python-Bibliothek, die in [ŘS10] vorgestellt wurde, für die Verarbeitung großer Textkorpora mit dem Vektorraummodell (VSM) im Bereich der natürlichen Sprachverarbeitung (NLP).
Diese Bibliothek eignet sich für Ähnlichkeitsaufgaben, bei denen Anfragen an das Vektorraummodell gestellt werden können.
Ein Problem, das sich dabei ergab, war, dass der Embedding-Vektor, der das verbalisierte Ziel repräsentiert, nicht Teil des Vektorraummodells war, das die Embeddings der Symbole enthält. In Gensim gestaltet sich das Hinzufügen oder Entfernen von Embeddings leider als umständlich.

Darüber hinaus konnte keine Funktion gefunden werden, die es ermöglicht, ein externes Embedding zu übergeben und darauf basierend Ähnlichkeiten zu berechnen. Eine Möglichkeit wäre gewesen, für jedes Ziel mit den Embeddings der Symbole ein neues Vektorspace-Modell zu erstellen.
Stattdessen wurde jedoch entschieden, in dieser Arbeit den Vergleich manuell mit der „`cosineSimilarity`“ Funktion aus PyTorch durchzuführen.
Ein Vorteil dabei ist, dass die Berechnung auf der Grafikkarte ausgeführt werden kann, was das Berechnen der Ähnlichkeiten beschleunigt.
Wie in Abschnitt 3.4 beschrieben, wurden 20 Prozent der erbrachten Beweise zu Analysezwecken zurückgehalten. Für diese Beweise wurden die entsprechenden Dateien, die die Axiome enthalten, zusammen mit den Beweiszielen ausgewählt. Diese Dateien wurden mit dem folgenden Eprover-Aufruf aufgerufen:

```
eprover „Pfad zum Ziel mit Axiomen“ -auto -silent
-print-statistics -proof-object -cpu-limit=20 -R
-outfile=„Pfad zum Speichern des Outputs“.
```

Die Verwendung des Aufrufs wird im weiteren Verlauf mit **w/o LM** (without language model), also ohne Verwendung des Sprachmodells, abgekürzt.
Der einzige Parameter, der im Vergleich zum Aufruf des Eprovers im Abschnitt 3.4 hinzugefügt wurde, ist `-print-statistics`. Dieser Parameter liefert zusätzliche Informationen, die durch `-silent` normalerweise unterdrückt, jedoch für spätere Vergleiche benötigt werden. Zudem wurde das Zeitlimit für die Bearbeitung jedes Beweises auf 20 Sekunden erhöht. Erstaunlicherweise konnten trotz der verlängerten Bearbeitungszeit von 15 auf 20 Sekunden und möglichst ähnlichen Rahmenbedingungen nicht alle zuvor erfolgreich geführten Beweise erneut erbracht werden.
Eine mögliche Erklärung hierfür ist, dass der Parameter `-print-statistics` das Sammeln von wesentlich mehr Informationen erfordert, was wiederum zu einer längeren Rechenzeit führt.
Für den Vergleich wurden die Dateien erneut mit dem folgenden veränderten Eprover-Aufruf aufgerufen:

```
eprover „Pfad zum Ziel mit Axiomen“ -silent
-print-statistics -proof-object -cpu-limit=20 -R
-x'(5*FunWeight(ConstPrio,1000,1,1,1,1,fw),1*FIFOWeight(ConstPrio))'
-outfile=„Pfad zum Speichern des Outputs“.
```

Die Verwendung des Aufrufs wird im weiteren Verlauf mit **w/ LM** (with language model), also mit Verwendung des Sprachmodells, abgekürzt.
Der Hauptunterschied zum vorherigen Aufruf besteht darin, dass der Parameter `-auto` nicht verwendet wurde, da dieser eine Reihe von Suchstrategien und Heuristiken ausprobiert, die den Vergleich beeinflussen könnten.
Stattdessen wird über den Parameter `-x` eine eigene Heuristik angegeben.
Diese lautet wie folgt:

```
-x'(5*FunWeight(ConstPrio,2,1000,1,1,1,fw),1*FIFOWeight(ConstPrio))'
```

Wie eine Heuristik definiert wird, wurde bereits im Kapitel 3.1 erläutert.
Die Funktionsweise der in dieser Arbeit verwendeten Heuristik lässt sich folgendermaßen beschreiben:
Für jede Gruppe von sechs Klauseln wird fünfmal die „FunWeight Funktion“ an-

gewandt. Diese Funktion evaluiert alle Klauseln, da durch die „Priority Function“ ConstPrio alle Klauseln ausgewählt werden.
Standardmäßig erhalten Funktionssymbole den Wert 1000, es sei denn, es wird explizit ein anderer Wert festgelegt, während Variablensymbole durchweg den Wert 1 bekommen. Maximale Terme gemäß der Termordnung sowie maximale und positive Literale innerhalb der Klauseln werden mit dem Wert 1 multipliziert. Die Bewertung der Symbole erfolgt durch das Sprachmodell, abhängig davon, wie gut sie zum Beweisziel passen.
Für jede Gruppe von sechs Klauseln wird die „FIFOWeight Funktion“ einmal angewendet. Diese Funktion bewertet die Klauseln nach dem FIFO-Prinzip (First In, First Out), wobei Klauseln, die zuletzt verarbeitet wurden, einen niedrigeren Wert erhalten.
Der Parameter „fw“ in der „FunWeight Funktion“, der die Symbole und deren Bewertungen enthält, ist ein String und wurde wie folgt erzeugt.
Zunächst wurde das zu beweisende Ziel nach der Methode verbalisiert, die bereits in Abschnitt 3.4 beschrieben wurde. Für das verbalisierte Ziel wurde dann das Embedding mittels des Sprachmodells berechnet. Dieser Embedding-Vektor wurde anschließend an das oben beschriebene Programm übergeben, welches die n am besten passenden und die k am schlechtesten passenden Symbole samt ihrer Ähnlichkeitswerte zurückgibt.
Da die Ähnlichkeitswerte mithilfe der Kosinusähnlichkeit berechnet wurden, liegen sie im Bereich zwischen 1 und -1. Ein Wert von 1 bedeutet, dass ein Symbol sehr gut zu einem zu beweisenden Ziel passt, während ein Wert von -1 bedeutet, dass ein Symbol sehr schlecht zu einem zu beweisenden Ziel passt.
Der endgültige Wert für die Bewertung des Symbols wurde wie folgt berechnet:

$$score = 1000 - (\text{Ähnlichkeitswert} \cdot 1000)$$

Der Berechnungsansatz für die Symbolbewertung basiert auf folgender Logik: Ein Symbol mit einem Ähnlichkeitswert zwischen 0 und 1, erhält einen niedrigeren „score“, da der Ähnlichkeitswert multipliziert mit 1000 von 1000 abgezogen wird. Dieser „score“ kann jedoch nicht unter 0 fallen, da der maximale Ähnlichkeitswert 1 ist und dieser nach der Multiplikation mit 1000 höchstens 1000 ergibt. Im Gegensatz dazu erhält ein Symbol, dessen Ähnlichkeitswert zwischen -1 und 0 liegt, einen erhöhten „score“, da der negative Ähnlichkeitswert multipliziert mit 1000 von 1000 abgezogen wird.
Der maximale „Score“ kann dabei 2000 nicht übersteigen, da der niedrigste mögliche Ähnlichkeitswert -1 ist und dieser, multipliziert mit 1000, zu einem minimalen Wert von -1000 führt. Dieser wird von 1000 abgezogen, dadurch ist der maximale „Score“ auf 2000 begrenzt.
Der Standardwert für nicht in der Heuristik angegebene Funktionsnamen wird auf 1000 festgelegt, da diese weder positiv noch negativ bewertet werden sollen.
Da die „Scores“ der einzelnen Symbole in der „FunWeight Funktion“, wie bereits in Abschnitt 3.1 beschrieben, durch Addition zur Bewertung der gesamten Klausel beitragen und Klauseln mit geringerer Bewertung bevorzugt vom Eprover ausgewählt werden, ergibt sich eine sinnvolle Logik für die Berechnung.

Beispiel 3.11. Als Beispiel sei `net(goal, symbol)` der vom Sprachmodell vorhergesagte Ähnlichkeitswert für ein Symbol. Angenommen, `goal` ist ein zu beweisendes Ziel, und die Symbole `f`, `g` und `h` sind die einzigen Symbole aus einer Klausel `C` mit den vorhergesagten Ähnlichkeitswerten `net(f, goal) = 0,8` und `net(h, goal) = -0,5`. Die berechneten „Scores“ für die Symbole ergeben sich nach der obigen Formel wie folgt: für das Symbol `f` 200 und für das Symbol `h` 1500.
In der beschriebenen Heuristik werden die Symbole `f` und `h` mit den berechneten „Scores“ angegeben. Vereinfacht dargestellt, wird dann von der „FunWeight Funktion“ die Bewertung der Klausel `C` wie folgt vorgenommen: die Werte werden addiert, also $200 + 1000 + 1500$. Dabei ist die 1000 der „Score“ für das Symbol `g`, da kein weiterer „Score“ für das Symbol `g` in der Heuristik angegeben ist, wird dieses mit dem „Default-Score“ bewertet.
Wie zu sehen ist, trägt ein gut zum `goal` passendes Symbol wie `f` zu einer besseren (negativeren) Bewertung der Klausel `C` bei, und ein schlecht passendes Symbol zum `goal` wie `h` führt zu einer schlechteren (positiveren) Bewertung der Klausel `C`. Das nicht spezifizierte Symbol `g` trägt weder zu einer besseren noch zu einer schlechteren Bewertung der Klausel `C` bei.

Die n am besten passenden und die k am schlechtesten passenden Symbole mit den jeweiligen „Scores“ wurden zu einem String zusammengesetzt für den Parameter „fw“, der folgendermaßen aufgebaut wurde:

$$\text{'symbolName}_1\text{:score}_1, \ldots, \text{symbolName}_{n+k}\text{:score}_{n+k}\text{'}$$

Ein Beispiel für den String „fw“ basierend auf den obigen Symbolen `f`, `g` und `h` wäre 'f:200, g:1000, h:1500'.
In der Anleitung des Eprover [Sch21] wird erwähnt, dass die „FunWeight Funktion“ ausschließlich Funktionssymbole akzeptiert. In dieser Arbeit wurden jedoch auch Prädikatensymbole bewertet. Durch gezielte Tests stellte sich heraus, dass auch diese angegeben werden können. Da der Eprover auf dem Superpositionskalkül basiert, welcher lediglich Gleichheits- und Ungleichheitsprädikate beinhaltet, könnte es sein, dass Prädikate intern als Funktionen interpretiert werden.
Im Folgenden wurden die beiden oben beschriebenen Aufrufe miteinander verglichen: einerseits der Aufruf mit dem Parameter `-auto` des Eprovers und andererseits der Aufruf mit der eigens implementierten Heuristik, welche die Ähnlichkeit von Symbolnamen zu einer Beweisaufgabe mittels eines Sprachmodells beurteilt.
Der `-auto` Parameter verwendet die Selektionstechnik SInE, um die Anzahl der Axiome in Adimen-SUMO zu reduzieren. Diese Selektionstechnik wird jedoch bei dem obigen Aufruf **w/ LM** mit der oben beschriebenen, eigens implementierten Heuristik nicht verwendet.
Um einen aussagekräftigen Vergleich durchzuführen, ist es essenziell, zunächst die zu vergleichenden Aspekte klar zu definieren und die Kernfragen festzulegen, die die Analyse beantworten soll. Ebenso wichtig ist es, Vergleichsmetriken festzulegen und die Daten zu benennen, welche für die Analyse verwendet wurden.

Vergleichsaspekte

Im Rahmen dieser Arbeit werden folgende Aspekte untersucht:

- **Effektivität des Beweisers:** Die Effektivität eines Beweisers misst, wie viele Beweise gefunden wurden. Die Bewertung der Effektivität ist wichtig, um zu bestimmen, ob der Einsatz der Heuristik praktisch sinnvoll ist. Wenig gefundene Beweise können darauf hinweisen, dass die Heuristik suboptimal ist und möglicherweise verbessert werden muss.
- **Arbeitslast des Beweisers:** Die Arbeitslast des Beweisers quantifiziert die Anzahl der Klauseln, die ein Beweiser verarbeiten muss, um zu einem Beweis zu gelangen. Dies bietet einen Einblick in den Aufwand, der für die Beweisführung erforderlich ist. Eine geringere Arbeitslast kann zu einer effizienteren und schnelleren Beweisfindung führen.

Kernfragen der Analyse

In der Analyse wurde versucht, folgende Fragen zu beantworten:

1. Inwiefern beeinflusst die eigens implementierte Heuristik die Effizienz und die Arbeitslast des Eprovers?
2. Welches ist die optimale Anzahl an best- und schlechtestpassenden Symbolen in der eigens implementierten Heuristik, um maximale Effektivität und minimale Arbeitsbelastung des Eprovers zu erreichen?
3. Kann eine Kombination des `-auto` Parameters mit der eigens implementierten Heuristik eine Verbesserung hinsichtlich Effektivität und Arbeitsbelastung des Eprovers erzielen?
4. Können Beweise für Probleme gefunden werden, die Eprover nur mit dem `-auto`-Modus nicht finden konnte?

Analysedaten

Der Vergleich basiert auf folgenden Daten: Ob ein Beweis mit dem Eprover gefunden wurde und, falls ein Beweis gefunden wurde, auf der Anzahl der vom Eprover verarbeiteten Klauseln.
Hierfür wurden die Ausgabedateien der entsprechenden Eprover-Aufrufe verwendet. Es wurde zudem sichergestellt, dass keine Beweisziele in den für den Vergleich herangezogenen Beweisen mehrfach auftraten. Ein doppeltes Beweisziel wurde identifiziert und aus den Ausgabedateien für die Analyse entfernt. Weiterhin wurde überprüft, ob Beweisziele, die in der Analyse verwendet wurden, auch im Training des Sprachmodells eingesetzt wurden. Dabei wurden vier solcher Beweisziele entdeckt und aus den Daten für den Vergleich entfernt. Die Daten sind in der Ausgabedatei des Eprovers wie folgt gekennzeichnet:

1. Ob ein Beweis gefunden wurde, ist in der Ausgabedatei des Eprovers durch `# Proof found!` gekennzeichnet.

2. Die Anzahl der vom Eprover verarbeiteten Klauseln, ist in der Ausgabedatei des Eprovers durch `# Processed clauses` markiert.

Es wurde ein Programm entwickelt, das die Informationen aus den Ausgabedateien des Eprovers extrahieren kann. Basierend auf den extrahierten Informationen wurden die beiden Aufrufe miteinander verglichen.
An dieser Stelle sei darauf hingewiesen, dass für den Vergleich potenziell noch deutlich mehr Informationen herangezogen werden könnten. Aus Gründen der Einfachheit wurde jedoch beschlossen, sich lediglich auf diese beiden Aspekte zu konzentrieren. Dies könnte ein Ansatzpunkt für weiterführende Forschungen sein.

Vergleichsmetriken

In der folgenden Tabelle 3.11 sind die Metriken aufgeführt, die für den Vergleich verwendet werden, um die Leistung des Eprovers zu beurteilen. Zudem wird begründet, warum diese Metriken ausgewählt wurden.

Statistisches Maß	**Begründung**
Anzahl der Beweisaufgaben, die für die Analyse verwendet wurden.	Dient als Referenzwert für die Anzahl der Beweise, die gefunden werden konnten.
Anzahl Beweise, die gefunden werden konnten.	Zeigt die Gesamtzahl der Beweise, die vom Beweiser erfolgreich gefunden wurden. Dies gibt Aufschluss über die Effektivität des Beweisers.
Anzahl verarbeiteten Klauseln.	Summe aller Klauseln, die während aller Beweisvorgänge verarbeitet wurden. Dies hilft, die Arbeitslast des Beweisers zu beurteilen.
Mittelwert der Anzahl verarbeiteten Klauseln pro Beweisvorgang.	Durchschnittliche Anzahl an Klauseln, die pro erfolgreichem Beweisvorgang verarbeitet wurden. Dieser Wert hilft, die Anzahl der verarbeiteten Klauseln besser zu verstehen und Beweisvorgänge zu verstehen.
Standardabweichung der Anzahl verarbeiteten Klauseln pro Beweisvorgang.	Maß für die Streuung oder Variabilität in der Anzahl der Klauseln pro Beweisvorgang. Hohe Werte deuten auf große Unterschiede zwischen den Beweisvorgängen hin.
Median der verarbeiteten Klauseln pro Beweisvorgang.	Der mittlere Wert der Klauseln, die pro Beweisvorgang verarbeitet wurden, wenn alle Werte der Größe nach geordnet sind. Zeigt die zentrale Tendenz der verarbeiteten Klauseln pro Beweisvorgang.
Verhältnis der Beweisvorgänge mit reduzierter Klauselanzahl unter verschiedenen Heuristiken.	Zeigt, wie sich die Anzahl der verarbeiteten Klauseln unter verschiedenen Heuristiken ändert. Es wird der Prozentsatz der Beweisvorgänge bestimmt, bei denen im Vergleich zu einer Basisheuristik eine Reduktion, Erhöhung oder keine Veränderung in der Anzahl der verarbeiteten Klauseln auftritt.
Durchschnittliche Reduktion der Klauseln pro Beweisvorgang in Prozent.	Gibt den durchschnittlichen Prozentsatz an, um den die Klauselanzahl pro Beweisvorgang reduziert wurde. Dies hilft, die Effizienz des Reduktionsprozesses zu bewerten.
Standardabweichung der Reduktion der Klauseln pro Beweisvorgang in Prozent.	Misst die Streuung in der Prozentsatzreduktion der Klauseln pro Beweisvorgang, was Aufschluss über die Konsistenz der Klauselreduktion gibt.
Median der Reduktion der Klauseln pro Beweisvorgang in Prozent.	Der mittlere Prozentsatz der Reduktion der Klauseln pro Beweisvorgang, angeordnet nach Größe. Zeigt die zentrale Tendenz der Reduktion pro Beweisvorgang.

Tabelle 3.11: Statistische Maße zur Bewertung der Leistung des Eprover.

Im Folgenden werden die Berechnungen der obigen statistischen Maße detailliert beschrieben.

Berechnung der Informationen über die Anzahl der Beweise

Zur Ermittlung der Anzahl der Beweisaufgaben wurden alle Ausgaben des Eprover iterativ durchlaufen und gezählt. Diese Kennzahl wird in der Arbeit als **Total proof tasks** bezeichnet und entspricht dem statistischen Maß der Anzahl der Beweisaufgaben, die für die Analyse verwendet wurden.
Der Eprover wurde in zwei Varianten aufgerufen welche in 3.6.1 beschrieben sind: einmal mit dem Aufruf **w/o LM** (ohne Sprachmodell) und einmal mit dem Aufruf **w/ LM** (mit Sprachmodell). Für jeden der beiden Aufrufe wurden Informationen darüber extrahiert, ob ein Beweis gefunden wurde, und die gefundenen Beweise wurden gezählt.
Die entsprechenden Kennzahlen werden als **Tasks solved w/o LM** und **Tasks solved w/ LM** abgekürzt. Diese repräsentieren das statistische Maß der Anzahl an Beweisen, die gefunden werden konnten.

Berechnung der Informationen über die verarbeiteten Klauseln

Der Eprover wurde in zwei Varianten aufgerufen welche in 3.6.1 beschrieben sind: einmal mit dem Aufruf **w/o LM** (ohne Sprachmodell) und einmal mit dem Aufruf **w/ LM** (mit Sprachmodell). Für alle Beweise, die mit beiden Aufrufvarianten gefunden wurden, wurden Informationen über die Anzahl der verarbeiteten Klauseln aus den jeweiligen Beweisvorgängen extrahiert.
Aus diesen Daten wurden die Gesamtzahl der Klauseln, der Durchschnitt, die Standardabweichung und der Median der verarbeiteten Klauseln pro Beweisvorgang berechnet.
Die berechneten Kennzahlen entsprechen den zuvor eingeführten statistischen Maßen, der Anzahl der verarbeiteten Klauseln, dem Mittelwert der verarbeiteten Klauseln pro Beweisvorgang, der Standardabweichung der verarbeiteten Klauseln pro Beweisvorgang und dem Median der verarbeiteten Klauseln pro Beweisvorgang.
Die Kennzahlen für den Eprover Aufruf **w/o LM** wurden in dieser Arbeit als **Total clauses w/o LM**, **Mean clauses w/o LM**, **Std deviation of clauses w/o LM** und **Median clauses w/o LM** abgekürzt. Entsprechend wurden die Kennzahlen für den Eprover Aufruf **w/ LM** als **Total clauses w/ LM**, **Mean clauses w/ LM**, **Std deviation of clauses w/ LM** und **Median clauses w/ LM** bezeichnet.

Berechnung der Informationen über die Reduktion der Klauseln

Es wurde einmal der Eprover ohne das Sprachmodell (mit dem Aufruf **w/o LM**) und einmal der Eprover mit dem Sprachmodell (mit dem Aufruf **w/ LM**) aufgerufen, mit den Aufrufen aus 3.6.1. Für die Fälle, in denen Beweise durch den

Aufruf mit beiden Varianten gefunden wurden, wurden Informationen bezüglich der Anzahl der verarbeiteten Klauseln aus den jeweiligen Beweisvorgängen extrahiert.
Von diesen wurde jeweils der Prozentsatz der gefundenen Beweisvorgänge bestimmt, bei denen die Anzahl der verarbeiteten Klauseln bei beiden Eprover-Aufrufen gleich war. Der Prozentsatz der Fälle, in denen die Anzahl der verarbeiteten Klauseln beim ersten Eprover-Aufruf geringer war als beim zweiten, und der Prozentsatz der Fälle, in denen die Anzahl der verarbeiteten Klauseln beim zweiten Eprover-Aufruf niedriger war als beim ersten, wurden ebenfalls bestimmt.
Die berechneten Kennzahlen entsprechen den zuvor eingeführten statistischen Maß: das Verhältnis der Beweisvorgänge mit reduzierter Klauselanzahl unter verschiedenen Heuristiken.
Die Kennzahlen werden als **% equal number of clauses**, **%fewer clauses w/o LM** und **%fewer clauses w/ LM** bezeichnet.
Ebenfalls wurde die prozentuale Reduktion der Klauseln berechnet, die wie folgt definiert wird:

$$\frac{\#\text{ Klauseln w/o LM} - \#\text{ Klauseln w/ LM}}{\#\text{ Klauseln w/o LM}} \cdot 100$$

Hierbei repräsentiert # **Klauseln w/o LM** die Anzahl der Klauseln, die benötigt wurden, wenn der Eprover mit dem Aufruf **w/o LM** aufgerufen wurde, und # **Klauseln w/ LM** die Anzahl der Klauseln für den Eprover-Aufruf **w/ LM**.
Von den prozentualen Reduktionen wurden dann der Durchschnitt, die Standardabweichung und der Median berechnet.
Diese Kennzahlen werden in der Arbeit als **Avg reduction of clauses**, **Std deviation of reduction**, und **Median of reduction** bezeichnet.
Ein Programm wurde entwickelt, das die Berechnungen aus den extrahierten Daten berechnet und ausgibt. Dabei kamen die Python-Bibliotheken NumPy und Pandas zum Einsatz.

3.6.2 Ergebnisse

Im folgenden Abschnitt werden die Ergebnisse präsentiert, die auf Basis der oben dargestellten Vergleichsdurchführung erzielt wurden.
Neben den in 3.6.1 definierten Bezeichnern für die statistischen Maße werden die in Tabelle 3.12 aufgeführten Abkürzungen zur Darstellung der Ergebnisse in den Tabellen verwendet.

Abkürzung	**Beschreibung**
Symb.	Abkürzung für logische Symbole, die verwendet wird, um die Symbole zu beschreiben, welche mittels der „FunWeight Funktion„ dem Eprover übergeben wurden.
Pos.	Abkürzung für positiv, die in Kombination mit der Abkürzung Symb. verwendet wird, um die Symbole zu beschreiben, die gut zu einer Beweisaufgabe passen, welche dann mittels der „FunWeight Funktion“ dem Eprover übergeben werden.
Neg.	Abkürzung für negativ, die in Kombination mit der Abkürzung Symb. verwendet wird, um die Symbole zu beschreiben, die nicht gut zu einer Beweisaufgabe passen, welche dann mittels der „FunWeight Funktion“ dem Eprover übergeben werden.
All.	Abkürzung für alle, die in Kombination mit der Abkürzung Symb. verwendet wird, um zu beschreiben, dass alle Symbole (Symbole, die gut zu einer Beweisaufgabe passen, Symbole, die schlecht zu einer Beweisaufgabe passen, und neutrale Symbole) mittels der „FunWeight Funktion„ dem Eprover übergeben werden.
-auto param.	Abkürzung für den `-auto` Parameter des Eprovers, um zu beschreiben, dass dieser bei dem Eprover-Aufruf **w/ LM** zusätzlich gesetzt wurde.

Tabelle 3.12: Abkürzungen und ihre Bedeutung für Tabellen.

Zunächst wurde der Eprover wie folgt aufgerufen:
`eprover ,,Pfad zum Ziel mit Axiomen‘‘ -silent -print-statistics -proof-object -cpu-limit=20 -R -outfile=,,Pfad zum Speichern des Outputs‘‘.`
Dies entspricht dem Aufruf **w/o LM** aus 3.6.1, allerdings mit dem Unterschied, dass der `-auto` Parameter des Eprovers nicht verwendet wurde und somit auch die SInE-Funktion ausgeschaltet blieb. Die Berechnungen wurden in Anlehnung an die in 3.6.1 beschriebenen Berechnungen durchgeführt.
Es ist anzumerken, dass nur die statistischen Maße berechnet wurden, die keinen Vergleich mit dem zweiten Aufruf **w/ LM** erfordern. Die Maße umfassen **Total proof tasks**, **Tasks solved w/o LM**, **Total clauses w/o LM**, **Mean clauses w/o LM**, **Std deviation of clauses w/o LM** und **Median clauses w/o LM**.
Bei der Berechnung ist ebenfalls wichtig, dass für die Berechnung nicht mehr die Beweisvorgänge der Beweise verwendet wurden, welche mit beiden Aufrufen **w/ LM** und **w/o LM** gefunden wurden. Dies lag daran, dass lediglich der eine oben genannte Aufruf getätigt wurde. Daher wurden für die Berechnung die Beweisvor-

gänge der Beweise verwendet, die mit diesem Aufruf gefunden wurden.
Die Ergebnisse sind in der nachfolgenden Tabelle 3.13 dargestellt.

Metric	**Value**
Total proof tasks	1845
Tasks solved w/o LM	968
Total clauses w/o LM	26058891
Mean clauses w/o LM	26920.34
Std deviation of clauses w/o LM	27252.11
Median clauses without LM	17118.0

Tabelle 3.13: Informationen über die Anzahl der gefundenen Beweise und der verarbeiteten Klauseln mit dem modifizierten Eprover-Aufruf **w/o LM** ohne den `-auto` Parameter.

Wie in Tabelle 3.13 dargestellt, wurde knapp die Hälfte der Beweise gefunden. Zudem weist der Mittelwert der verarbeiteten Klauseln einen hohen Wert auf, was auf Schwankungen in der Anzahl der verarbeiteten Klauseln bei verschiedenen Beweisprozessen hindeutet. Dies wird ebenfalls durch die hohe Standardabweichung bestätigt. Der Median der verarbeiteten Klauseln beträgt etwa 17118, was auf die durchschnittliche Anzahl an Klauseln hinweist, die pro Beweis benötigt wird.
Wichtig: In den nachfolgenden Tabellen ist zu beachten, dass sich die Werte für die verarbeiteten Klauseln und die Reduktionen der Klauseln auch bei der Verwendung des **w/o LM** Eprover-Aufrufs stetig ändern. Dies ist auf die in Abschnitt 3.6.1 beschriebene Berechnungsmethode zurückzuführen, welche die Beweisprozesse analysiert, bei denen Beweise sowohl mit dem **w/o LM** Eprover-Aufruf als auch mit dem **w/ LM** Eprover-Aufruf gefunden wurden.
Als nächstes wurde die Anzahl der am besten passenden Symbole, die dem Eprover übergeben wurden, schrittweise um jeweils fünf erhöht. Dies bedeutet, dass bei jedem **w/ LM**-Aufruf des Eprovers, die „FunWeight Funktion“ um fünf zusätzliche, am besten passende Symbole erweitert wurde.
Das Ergebnis ist in der folgenden Tabelle 3.14 zu sehen.

Metric	**Pos. 5 symb.**	**Pos. 10 symb.**	**Pos. 15 symb.**	**Pos. 20 symb.**	**Pos. 25 symb.**	**Pos. 30 symb.**
Total proof tasks	1845	1845	1845	1845	1845	1845
Tasks solved w/o LM	1522	1522	1522	1522	1522	1522
Tasks solved w/ LM	866	891	900	908	913	919
Total clauses w/o LM	16787253	17497473	17739348	17895204	18004990	18212956
Total clauses w/ LM	13153857	13357428	13290360	13248522	13253272	13364779
Mean clauses w/o LM	19843.09	20135.18	20227.31	20266.37	20298.75	20395.25
Mean clauses w/ LM	15548.29	15371.03	15154.34	15003.99	14941.68	14966.16
Std deviation of clauses w/o LM	14003.67	14052.91	14036.67	13999.38	14057.01	14055.33
Std deviation of clauses w/ LM	9959.12	9914.45	9704.91	9546.93	9652.46	9847.29
Median clauses w/o LM	18100.5	18215.0	18225.0	18235.0	18235.0	18263.0
Median clauses w/ LM	16622.0	16628.0	16629.0	16634.0	16640.0	16645.0
% fewer clauses w/o LM	32.62%	30.15%	28.39%	28.65%	28.41%	28.22%
% fewer clauses w/ LM	62.88%	65.36%	67.16%	66.93%	67.19%	67.41%
% equal number of clauses	4.49%	4.49%	4.45%	4.42%	4.40%	4.37%
Avg reduction of clauses	-1862.66%	-1460.63%	-1407.96%	-1316.43%	-1197.72%	-1190.91%
Std deviation of reduction	14280.97	12079.45	11755.17	11500.65	10958.92	10933.81
Median of reduction	21.48%	22.50%	23.10%	23.06%	23.37%	23.56%

Tabelle 3.14: Vergleich des **w/o LM**-Aufrufs des Eprover mit dem **w/LM**-Aufruf des Eprovers durch Anpassung der Anzahl der am besten passenden Symbolnamen.

Basierend auf den in Tabelle 3.14 dargestellten Daten lässt sich Folgendes hinsichtlich des Vergleichs der Menge der übergebenen Symbole feststellen:
Die Anzahl der gefundenen Beweise scheint zuzunehmen, je mehr Symbole dem Beweiser bei der Nutzung des Sprachmodells zur Verfügung gestellt werden, was auf eine gesteigerte Effektivität des Beweisers hinweist. Zudem zeigt sich ein leichter Anstieg des Medians der prozentualen Reduktion der Klauseln, was darauf schließen lässt, dass pro Beweisvorgang weniger Klauseln verarbeitet werden müssen, was eine geringere Arbeitslast für den Eprover bedeutet. Auffällig ist auch, dass der Anteil der Beweisprozesse, bei denen unter Einsatz des Sprachmodells weniger Klauseln benötigt wurden, steigt, was darauf hindeutet, dass sich der Einsatz der Heuristik bei einer zunehmenden Zahl von Beweisen als vorteilhaft erweist.
Aufgrund des begrenzten zeitlichen Rahmens der Bachelorarbeit wurde, um noch weitere interessante Aspekte zu untersuchen, die Anzahl der am besten passenden Symbole nicht weiter erhöht. Dies bietet jedoch einen Ansatzpunkt für zukünftige Forschungen. Es zeichnet sich bereits ein Trend ab: Die Ergänzung von zusätzlichen, zur Beweisaufgabe passenden Symbolen führt dazu, dass mehr Beweise gefunden werden. Zudem reduziert sich die Anzahl der verarbeiteten Klauseln, und die prozentuale Reduktion im Vergleich zum **w/o LM**-Aufruf des Eprover steigt. Ob es also einen optimalen Wert für die Anzahl der am besten passenden Symbole gibt, bleibt im Rahmen dieser Arbeit offen.
Es ist ebenfalls von Bedeutung zu analysieren, inwieweit Symbole, die nicht gut zu einer Beweisaufgabe passen, den Prozess des Beweisführens verändern.
Die vorherigen Daten haben gezeigt, dass die meisten Beweise gefunden wurden, als die 30 am höchsten bewerteten Symbole dem Eprover zur Verfügung gestellt wurden. Daher wurden in der nachfolgenden Analyse sowohl die 30 am besten bewerteten als auch die 10 am schlechtesten bewerteten Symbole dem Eprover übergeben.
Auf diese Weise soll analysiert werden, welchen Effekt die zusätzliche Übergabe der 10 am schlechtesten bewerteten Symbole hat.
Ebenfalls wurden zur Analyse einmal alle Symbole (die am besten zur Beweisaufgabe passenden Symbole, die am schlechtesten zur Beweisaufgabe passenden Symbole und neutrale Symbole) dem Eprover übergeben, um eine umfassende Übersicht darüber zu erhalten, wie sich das Übergeben von allen negativ bewerteten Symbolen, allen positiv bewerteten Symbolen und allen neutral bewerteten Symbolen auf den Beweisprozess auswirkt.
Die Ergebnisse dieser Analysen sind in der folgenden Tabelle 3.15 aufgeführt.
Um eine bessere Vergleichbarkeit zu gewährleisten, wurden die Daten für die Übergabe der 30 am höchsten bewerteten Symbole an den Eprover aus Tabelle 3.14 in die aktuelle Tabelle integriert.

Metric	**Pos. 30 symb.**	**Pos. 30, Neg. 10 symb.**	**All symb.**
Total proof tasks	1845	1845	1845
Tasks solved w/o LM	1522	1522	1522
Tasks solved w/ LM	919	920	1039
Total clauses w/o LM	18212956	18258508	21516636
Total clauses w/ LM	13364779	13364343	15098855
Mean clauses w/o LM	20395.25	20423.39	21367.07
Mean clauses w/ LM	14966.16	14948.93	14993.90
Std deviation of clauses w/o LM	14055.33	14072.62	13855.97
Std deviation of clauses w/ LM	9847.29	9828.58	13378.00
Median clauses w/o LM	18263.0	18265.5	20269.0
Median clauses w/ LM	16645.0	16593.5	13040.0
% fewer clauses w/o LM	28.22%	28.08%	25.72%
% fewer clauses w/ LM	67.41%	67.56%	70.51%
% equal number of clauses	4.37%	4.36%	3.77%
Avg reduction of clauses	-1190.91%	-1189.35%	-439.02%
Std deviation of reduction	10933.81	10927.79	4624.80
Median of reduction	23.56%	23.71%	34.77%

Tabelle 3.15: Vergleich des **w/o LM**-Aufrufs des Eprovers mit dem **w/ LM**-Aufruf des Eprovers: mit den 30 am besten passenden Symbolnamen, den 30 am besten passenden Symbolen und den 10 am schlechtesten passenden Symbolen sowie mit allen Symbolen.

Wie in Tabelle 3.15 dargestellt, führte das zusätzliche Übergeben der zehn am wenigsten passenden Symbole an den Eprover, im Vergleich zur Übergabe der 30 am besten passenden Symbole an den Eprover, zu kaum merklichen Veränderungen. Die Ergebnisse blieben sehr ähnlich. Insgesamt wurden jedoch etwas weniger Klauseln pro Beweis benötigt, und es konnte ein zusätzlicher Beweis unter Zuhilfenahme der zehn am schlechtesten passenden Symbole gefunden werden.
Das Mitgeben aller Symbole (am besten passende Symbole, am schlechtesten passende Symbole und neutral bewertete Symbole) an den Eprover beim Aufruf hatte einen großen Einfluss. Insbesondere stieg die Anzahl der gefundenen Beweise.
Der Anteil der Beweisprozesse, bei denen durch den Einsatz des Sprachmodells weniger Klauseln verarbeitet wurden, ist gestiegen.
Ebenfalls zeigte sich eine Erhöhung bei der durchschnittlichen prozentualen Reduktion der Anzahl der verarbeiteten Klauseln.
Es bleibt offen, ob es einen optimalen Wert für die Anzahl der am besten und am schlechtesten zur Beweisaufgabe passenden Symbole gibt, welche an den Eprover übergeben werden, um die höchste Effektivität (gemessen an der Anzahl gefundener Beweise) und die niedrigste Arbeitslast (gemessen an der Anzahl verarbeiteter Klauseln pro Beweisprozess) zu erreichen.

Die vorherigen Ergebnisse deuten jedoch darauf hin, dass mit der Zunahme der dem Eprover übergebenen Symbole die Effektivität steigt und die Arbeitslast pro Beweis abnimmt.
Weitere Forschung ist notwendig, um präzisere Ergebnisse zu erhalten und ein tieferes Verständnis des Einflusses der Anzahl übergebener Symbole an den Eprover auf den Beweisprozess zu entwickeln.
Auffällig an den obigen Daten ist, dass sich die Effektivität (gemessen an der Anzahl der gefundenen Beweise) bei der Verwendung des Sprachmodells insgesamt reduziert hat. Es erscheint denkbar, dass eine Kombination der eigens implementierten Heuristik mit dem `-auto`-Parameter des Eprovers zu besseren Ergebnissen führen könnte.
Dies liegt daran, dass durch die Verwendung des `-auto`-Parameters des Eprovers die SInE-Selektion verwendet wird. Dies führt zu einer Reduktion der Axiome und dadurch potenziell zu mehr gefundenen Beweisen. Diese Möglichkeit wurde im Folgenden untersucht.
Hierfür wurde der **w/ LM**-Aufruf des Eprover aus 3.6.1, welcher die Heuristik definiert, wie folgt abgeändert:

```
eprover „Pfad zum Ziel mit Axiomen“ -auto -silent
-print-statistics -proof-object -cpu-limit=20 -R
-x'(5*FunWeight(ConstPrio,1000,1,1,1,1,fw),1*FIFOWeight(ConstPrio))'
-outfile=„Pfad zum Speichern des Outputs“
```

Zusätzlich zur definierten Heuristik wurde der Parameter `-auto` des Eprovers verwendet.
Die obigen Daten zeigen, dass die besten Ergebnisse erzielt wurden, wenn alle Symbole an den Eprover übergeben wurden. Für die Analyse wurden daher alle Symbole dem Eprover übergeben. Dieser wurde mit dem oben beschriebenen, modifizierten Aufruf aufgerufen.
Im der folgenden Tabelle 3.16 ist das Ergebnis des Aufrufs abgebildet. Die Verwendung des modifizierten Aufrufs ist in der Tabelle mit **-auto param.** gekennzeichnet.
Zum Vergleich wurden zusätzlich die Daten des vorherigen Aufrufs, bei dem alle Symbole dem Eprover übergeben wurden, aus der Tabelle 3.15 in die Tabelle 3.16 übernommen.

Metric	**All Sybm.**	**All Sybm., -auto param.**
Total proof tasks	1845	1845
Tasks solved w/o LM	1522	1522
Tasks solved w/ LM	1039	1156
Total clauses w/o LM	21516636	24877759
Total clauses w/ LM	15098855	14297341
Mean clauses w/o LM	21367.07	22432.61
Mean clauses w/ LM	14993.90	12892.10
Std deviation of clauses w/o LM	13855.97	13795.69
Std deviation of clauses w/ LM	13378.00	8064.45
Median clauses w/o LM	20269.0	23865.0
Median clauses w/ LM	13040.0	13891.0
% fewer clauses w/o LM	25.72%	19.03%
% fewer clauses w/ LM	70.51%	77.55%
% equal number of clauses	3.77%	3.43%
Avg reduction of clauses	-439.02%	-380.53%
Std deviation of reduction	4624.80	4360.08
Median of reduction	34.77%	40.71%

Tabelle 3.16: Vergleich des **w/o LM**-Aufrufs des Eprovers mit dem **w/ LM**-Aufruf des Eprovers und des **w/o LM**-Aufrufs mit dem modifizierten **w/ LM**-Aufruf, ergänzt um den zusätzlichen **-auto**-Parameter des Eprovers, unter Einbeziehung aller Symbole.

Wie in Tabelle 3.16 ersichtlich, wurde die Anzahl der gefundenen Beweise unter Zuhilfenahme des **-auto**-Parameters des Eprovers erhöht.
Auch die Anzahl der verarbeiteten Klauseln und der verarbeiteten Klauseln pro Beweis hat sich unter Zuhilfenahme des **-auto**-Parameters des Eprovers reduziert. Ebenfalls hat sich der Median der durchschnittlichen Anzahl der Reduktionen unter Zuhilfenahme des **-auto**-Parameters des Eprovers erhöht, sodass tendenziell etwa 40 % weniger Klauseln pro Beweis verarbeitet werden müssen.
Zudem ist durch die zusätzliche Verwendung des **-auto**-Parameters des Eprovers der Anteil der Beweisprozesse gestiegen, bei denen der Einsatz des Sprachmodells die Anzahl der verarbeiteten Klauseln reduziert hat.
Ein weiterer Ansatz für zukünftige Forschung könnte darin bestehen, die Auswirkungen der Kombination der in dieser Arbeit eigens implementierten Heuristik mit anderen Heuristiken zu untersuchen.
Auffällig an allen bisherigen Daten ist, dass die Standardabweichung bei der Anzahl der verarbeiteten Klauseln sowohl mit als auch ohne Sprachmodell hoch ist. Dies deutet darauf hin, dass zur Lösung der Beweise sehr unterschiedliche Mengen an Klauseln benötigt wurden.
Der Median der verarbeiteten Klauseln weicht sowohl bei der Verwendung des Sprachmodells als auch ohne dieses leicht vom Durchschnitt der verarbeiteten Klauseln ab. Dies könnte darauf hindeuten, dass es Ausreißer gibt, also Bewei-

se, für die entweder sehr wenige oder sehr viele Klauseln benötigt wurden. Dies ist jedoch positiv zu bewerten, da es darauf hindeutet, dass die analysierten Daten eine Vielfalt aufweisen, die der realen Situation bei der Durchführung von Beweisen entspricht.
Auffällig ist auch, dass der Median der prozentualen Reduktion der verarbeiteten Klauseln stark von der durchschnittlichen prozentualen Reduktion der verarbeiteten Klauseln abweicht. Zudem ist die Standardabweichung der prozentualen Reduktion der verarbeiteten Klauseln hoch. Dies deutet auf große Ausreißer hin.
Im Folgenden wird eine grafische Darstellung der erzielten Ergebnisse des Eprover-Aufrufs gezeigt. Dabei wurden alle Symbole dem Eprover übergeben und der **-auto**-Parameter des Eprover wurde zusätzlich wie oben verwendet. Dieser Aufruf erzielte basierend auf den obigen Daten die besten Ergebnisse.

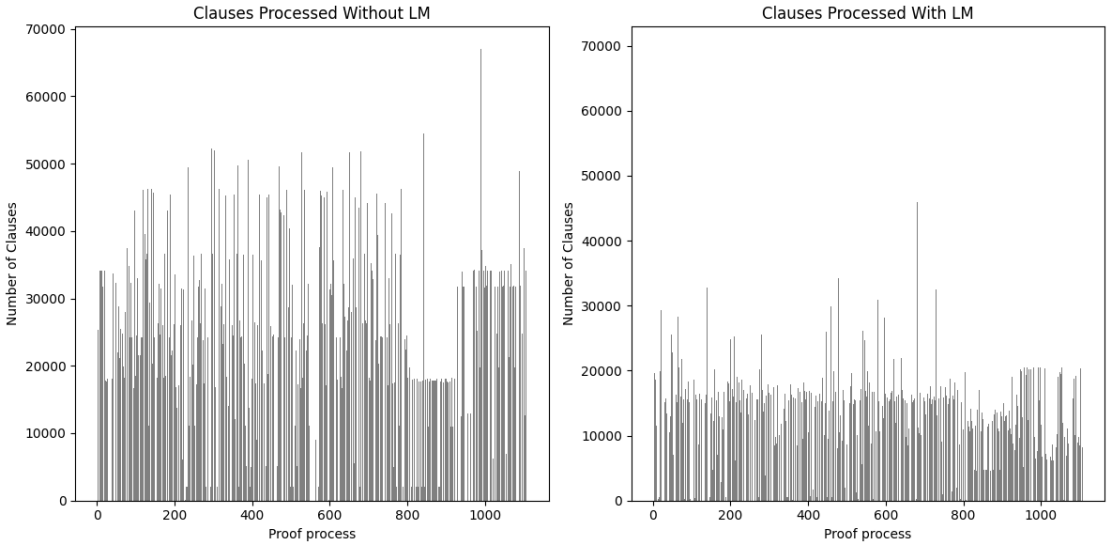


Abbildung 3.24: Vergleich der Anzahl verarbeiteter Klausuren ohne den Einsatz des Sprachmodells (links) und mit dem Einsatz des Sprachmodells (rechts).

Wie in Abbildung 3.24 zu sehen ist, bestätigt sich die zuvor geäußerte Vermutung, dass es einige Beweisprozesse mit einer sehr geringen Anzahl verarbeiteter Klauseln und andere mit einer sehr hohen Anzahl verarbeiteter Klauseln gibt. Insgesamt zeigt der Vergleich der Anzahl verarbeiteter Klauseln deutlich, dass bei Verwendung des Sprachmodells wesentlich weniger Klauseln verarbeitet werden mussten. Besonders stark reduziert haben sich die Beweisprozesse, die zuvor eine große Anzahl an Klauseln erforderten. Diese scheinen nun deutlich weniger Klauseln zu benötigen.

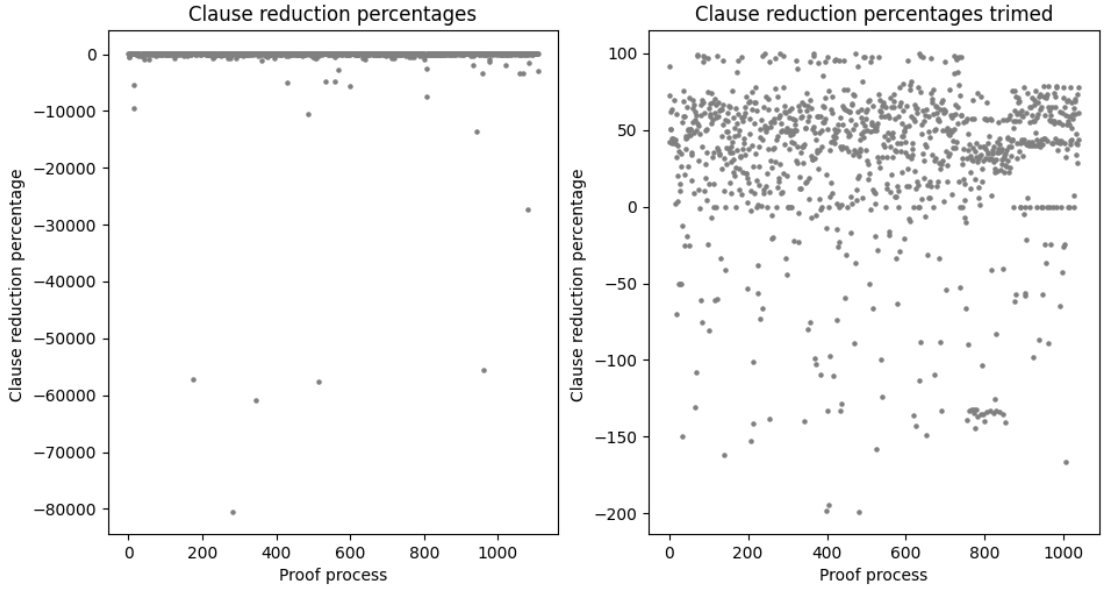


Abbildung 3.25: Reduktionsprozentsätze der verarbeiteten Klauseln ohne Entfernung der Ausreißer (links) und mit Entfernung der Ausreißer (rechts).

Wie in Abbildung 3.25 zu sehen ist, hat sich die obige Vermutung bestätigt: Es gibt einige Ausreißer, bei denen sich die Anzahl der Klauseln extrem erhöht hat. Diese Erhöhungen sind so stark, dass andere Reduktionen kaum noch erkennbar sind. Aus diesem Grund wurden in der rechten Abbildung die 70 negativsten Werte entfernt, um einen besseren Überblick über die Reduktionen zu erhalten. In dieser Abbildung wird deutlich, dass in den meisten Beweisprozessen die Anzahl der Klauseln reduziert wurde. Es gibt jedoch einige Beweisprozesse, bei denen sich die Anzahl der Klauseln extrem stark erhöht hat.

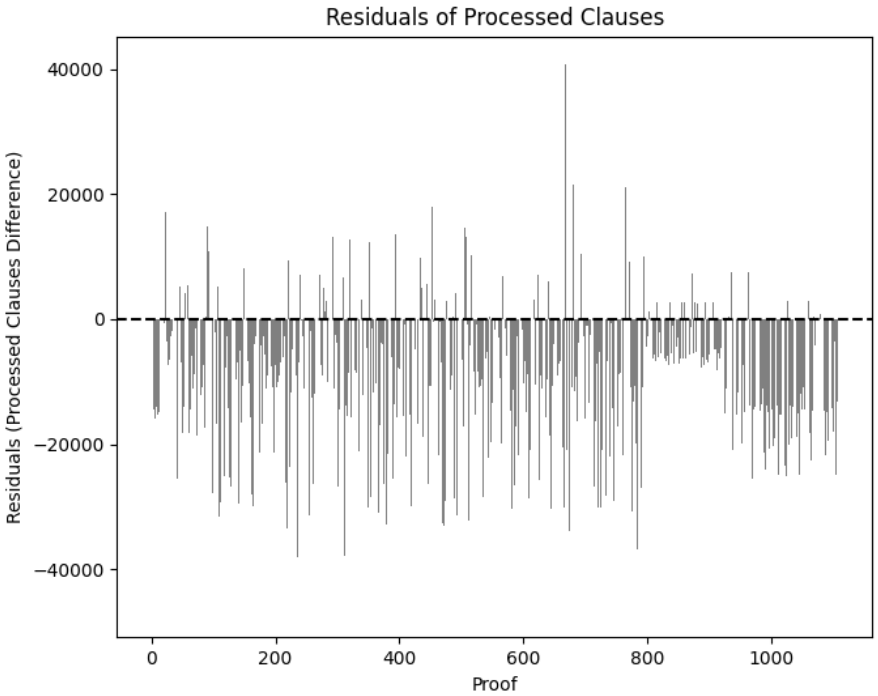


Abbildung 3.26: Residuen zwischen den verarbeiteten Klauseln. Mit Residuen ist die Anzahl der verarbeiteten Klauseln mit Verwendung des Sprachmodells subtrahiert von der Anzahl der verarbeiteten Klauseln ohne Verwendung des Sprachmodells gemeint. Ein negativer Wert bedeutet, dass sich die Anzahl der Klauseln für einen Beweisprozess reduziert hat, und ein positiver, dass sich die Anzahl der Klauseln für einen Beweisprozess erhöht hat.

Wie in Abbildung 3.26 zu sehen ist, sind die meisten Werte negativ. Das bedeutet, dass sich in den meisten Fällen die Anzahl der verarbeiteten Klauseln deutlich verringert hat. Lediglich in einigen wenigen Fällen hat sie sich erhöht.

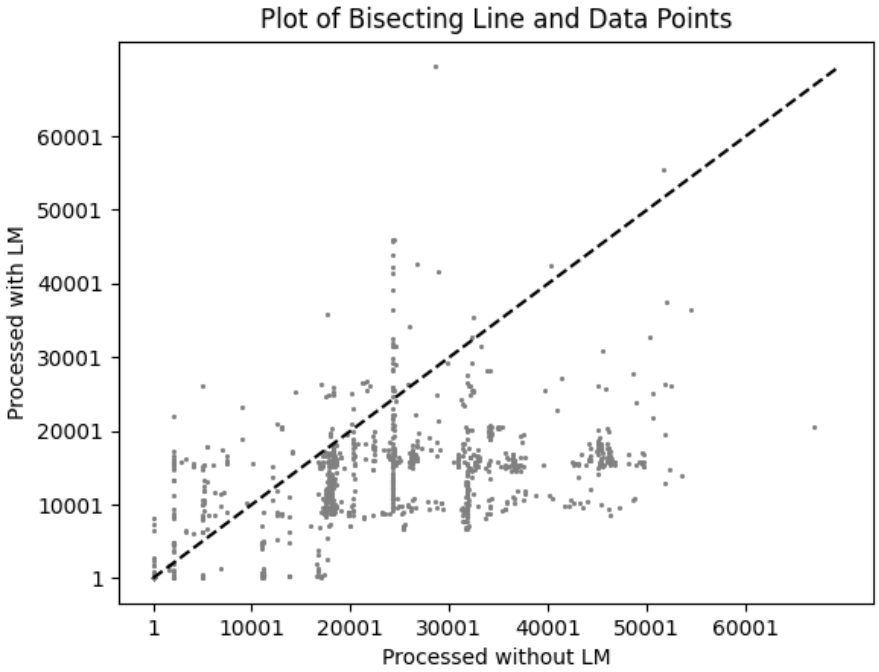


Abbildung 3.27: Darstellung der verarbeiteten Klauseln mit und ohne Verwendung des Sprachmodells.

Abbildung 3.27 liefert einen umfassenden Überblick über die Ergebnisse des Vergleichs. Sie ähnelt Abbildung 3.24, bietet jedoch durch die Zusammenführung beider Darstellungen in einem einzigen Bild eine leicht veränderte Perspektive. Dieser Ansatz bietet den Vorteil, dass die Balken der beiden Einzelabbildungen nicht gedanklich nebeneinandergestellt werden müssen, um die Anzahl der verarbeiteten Klauseln zu vergleichen.
Bei allen Werten unterhalb der Winkelhalbierenden ist die Anzahl der verarbeiteten Klauseln geringer. Bei allen Werten oberhalb der Winkelhalbierenden ist diese Anzahl höher, und auf der Winkelhalbierenden sind sie gleich.
Der Abstand zur Winkelhalbierenden zeigt, wie stark sich die Anzahl der verarbeiteten Klauseln erhöht oder verringert hat. Wie in der Abbildung zu sehen ist, hat sich in der Mehrzahl der Fälle die Anzahl der verarbeiteten Klauseln verringert, da sie sich unterhalb der Winkelhalbierenden befinden.
Eine weitere interessante Fragestellung ist, ob mit Hilfe der eigens entwickelten Heuristik Beweise gefunden werden können, die zuvor nicht erbracht werden konnten. Im Abschnitt 3.4 wurden die Ziele der bisher nicht erbrachten Beweise für Analysezwecke aufgehoben. Diese wurden erneut mit dem folgenden Befehl versucht zu lösen:

```
eprover „Pfad zum Ziel mit Axiomen“- auto -silent
-print-statistics -proof-object -cpu-limit=15 -R
-x'(5*FunWeight(ConstPrio,1000,1,1,1,1,fw),1*FIFOWeight(ConstPrio))'
-outfile=„Pfad zum Speichern des Outputs“.
```

Es ist wichtig zu beachten, dass beim Aufruf der `-auto`-Parameter zusammen

mit der eigens implementierten Heuristik verwendet wurde und alle Symbolnamen mitgegeben wurden. Dieser Aufruf erzielte nach den vorherigen Daten die besten Ergebnisse hinsichtlich Effizienz und Arbeitslast des Beweisers.
Ebenso wichtig ist, dass der Parameter `-cpu-limit` wieder auf 15 Sekunden eingestellt wurde, da im Abschnitt 3.4 versucht wurde, die Beweise mit einem CPU-Limit von 15 Sekunden zu finden.
Insgesamt konnten mithilfe der eigens implementierten Heuristik unter möglichst gleichen Bedingungen von 5749 nicht gefundenen Beweisen aus Applying CWA 146 neue Beweise gefunden werden, und von den 26216 nicht gefundenen Beweisen aus Adimen-SUMO konnten 2276 neue Beweise gefunden werden. Insgesamt konnten so aus 31965 nicht gefundenen Beweisen 2422 neue Beweise gefunden werden.
Somit wurden ungefähr 7,58 neue Beweise im Vergleich zu den zuvor nicht entdeckten Beweisen gefunden.

3.7 Diskussion der Ergebnisse

Die Analyse und Präsentation der Ergebnisse des Vergleichs zwischen dem Eprover-Aufruf mit der eigens implementierten Heuristik und dem Eprover-Aufruf ohne Heuristik in Kombination mit dem `-auto`-Parameter liefert interessante Erkenntnisse.
Es wurde festgestellt, dass die Anzahl der gefundenen Beweisaufgaben mit dem Einsatz des Sprachmodells ansteigt, je mehr Symbole dem Beweiser zur Verfügung gestellt werden. Allerdings hat sich die Effektivität des Beweisers insgesamt durch den Einsatz des Sprachmodells deutlich verringert.
Die Ergebnisse zeigen, dass das übergeben von mehr gut zur Beweisaufgabe passenden Symbolen (bis zu 30 Symbole) an den Eprover zu einer Erhöhung der Anzahl gelöster Beweisaufgaben und einer Reduktion der verarbeiteten Klauseln führt.
Die Einbeziehung von zehn schlecht passenden Symbolen hat zu einer minimalen Veränderung der Ergebnisse geführt. Es zeigt sich, dass diese Symbole die Effektivität und Arbeitslast des Beweisers nur geringfügig beeinflussen.
Die Übergabe aller verfügbaren Symbole an den Eprover hat die Anzahl der gefundenen Beweise erhöht und die Anzahl der verarbeiteten Klauseln weiter reduziert. Dies deutet darauf hin, dass eine umfassendere Symbolauswahl dem Eprover nützlich sein kann.
Die Kombination des Sprachmodells mit dem `-auto` Parameter des Eprovers hat die Anzahl der gefundenen Beweise weiter erhöht. Auch die Anzahl der verarbeiteten Klauseln pro Beweis hat sich signifikant reduziert. Der Median der durchschnittlichen Reduktion der Klauseln stieg auf etwa 40 %, was auf eine erheblich geringere Arbeitslast des Eprovers hinweist.
Die hohe Standardabweichung der verarbeiteten Klauseln sowohl mit als auch ohne Sprachmodell deutet darauf hin, dass es große Unterschiede in der benötigten Klauselmenge zur Lösung der Beweise gibt.
Der Median der prozentualen Reduktion weicht deutlich vom Durchschnitt der prozentualen Reduktion ab. Dies deutet darauf hin, dass bei einigen Beweisvorgängen

durch die Nutzung des Sprachmodells die Anzahl der verarbeiteten Klauseln stark zugenommen hat.
Diese Beobachtungen stimmen mit den grafischen Darstellungen überein.
Die grafischen Darstellungen bestätigen, dass bei Verwendung des Sprachmodells im Durchschnitt weniger Klauseln verarbeitet werden mussten. Insbesondere bei Beweisprozessen, die zuvor eine hohe Anzahl an Klauseln erforderten, wurde die Anzahl der verarbeiteten Klauseln stark reduziert.
Die Reduktionsprozentsätze zeigen, dass in den meisten Fällen die Anzahl der Klauseln reduziert wurde.
Die Darstellung der Residuen zeigt, dass in den meisten Fällen die Anzahl der verarbeiteten Klauseln verringert wurde, was auf eine Verbesserung des Beweisprozesses durch den Einsatz des Sprachmodells hinweist.
Ebenfalls konnten 2422 neue Beweise gefunden werden, die durch den Einsatz des `-auto` Parameter des Epvoer ohne den Einsatz des Sprachmodells nicht gefunden wurden.
Zukünftige Forschung könnte untersuchen, welche Kombinationen von Heuristiken die besten Ergebnisse liefern, um sowohl die Effektivität als auch die Effizienz des Beweisprozesses weiter zu verbessern.
Weitere Analysen könnten sich ebenfalls auf die Untersuchung der Ausreißer konzentrieren, um die Gründe für die extreme Variabilität in der Anzahl der verarbeiteten Klauseln besser zu verstehen.
Interessant für weitere zukünftige Arbeiten wäre es auch zu untersuchen, welche Beweise verloren gegangen sind und welche neu gefunden wurden, um diese Probleme genauer zu analysieren.
Zusammenfassend lässt sich sagen, dass gezeigt wurde, dass die Effektivität des Beweisers durch den Einsatz des Sprachmodells zwar verringert wird, gleichzeitig jedoch die Anzahl der verarbeiteten Klauseln erheblich reduziert werden konnte. Ebenfalls konnten durch den Einsatz des Sprachmodells neue Beweise gefunden werden, die zuvor nicht gefunden wurden.
Es ist daher möglich, aus vorangegangenen Beweisen zu lernen, indem die Bedeutung der Symbolnamen und der zu beweisenden Ziele berücksichtigt wird und dadurch eine Heuristik zu entwickeln, die einen Mehrwert liefert.
Dennoch besteht weiterhin ein großer Forschungsbedarf auf diesem Gebiet.

3.7.1 Ressourcen und Rahmenbedingungen für die Reproduzierbarkeit

Dieses Kapitel erläutert die Ressourcen und Rahmenbedingungen, die für die Durchführung dieser Arbeit notwendig waren. Es wird beschrieben, wo die in dieser Arbeit entwickelte Software und die erzeugten Daten zu finden sind.

Eingesetzte Hardware und Software

Im Rahmen dieser Arbeit wurde die folgende Hardware und Software eingesetzt:

1. **Berechnung der Trainingsdaten:**

- **Hardware:** NVIDIA GeForce GTX 950 mit 2 GB VRAM, AMD Ryzen 5 3600 AMD64 Family 23 Model 113 Stepping 0 AuthenticAMD 16 GB DDR4 RAM mit 1333 MHz (verfügbarer Arbeitsspeicher: 15.95 GB)
- **Software:** Ubuntu 20.04, Eprover 2.6

2. **Vergleichsanalyse der Verlustfunktionen:**
   - **Hardware:** Identisch zur Berechnung der Trainingsdaten
   - **Software:** Windows 11, PyTorch Version 2.2.0+cu118, Cuda Version 11.8
3. **Training des Sprachmodells mit dem vollständigen Trainingsdatensatz:**
   - **Hardware:** NVIDIA GeForce RTX 3060 mit 12 GB VRAM, AMD Ryzen 5 3600 AMD64 Family 23 Model 113 Stepping 0 AuthenticAMD, 16 GB DDR4 RAM mit 3200 MHz (verfügbarer Arbeitsspeicher: 15.95 GB)
   - **Software:** Windows 10, PyTorch Version 2.1.0+cu118, Cuda Version 11.8
4. **Training des Sprachmodells mit durch Undersampling angepassten Trainingsdatensätzen:**
   - **Hardware:** Identisch zur Berechnung der Trainingsdaten
   - **Software:** Identisch zur Vergleichsanalyse
5. **Evaluation:**
   - **Hardware:** Identisch zur Berechnung der Trainingsdaten
   - **Software:** Ubuntu 20.04, Eprover 2.6, PyTorch Version 2.2.0+cu118, Cuda Version 11.8

Ergebnisqualität

Um die Vergleichbarkeit der Ergebnisse zu optimieren, wurden kritische Berechnungen vorzugsweise nachts durchgeführt, um Beeinträchtigungen durch andere Systembelastungen zu minimieren. Dies sollte jedoch bereits durch die angegebene Rechenzeit beim Aufruf des Eprovers gewährleistet sein.

Verfügbarkeit von Software und Daten

Software: Die im Rahmen dieser Arbeit entwickelte Software ist öffentlich im GitHub-Repository unter `https://github.com/kossmanf/bachelorThesis.git` verfügbar.
Ablage von großen Dateien: Große Dateien wie Trainingsdaten, Testdaten, erzeugte Axiome mit dem Eprover sowie Beweise und Testaxiome sind im folgenden Google Drive-Ordner öffentlich zugänglich: `https://drive.google.com/drive/u/0/folders/1CdyYpplWQSY6nyqx5S7C_CnwbHbhjrnu`.

4

Zusammenfassung und Ausblick

Das Verständnis und die Weiterentwicklung von automatischen Schlussfolgern, also der Schlussfolgerungsfähigkeit, sind zentrale Aspekte in der Forschung der künstlichen Intelligenz und spielen eine wesentliche Rolle in der Informatik.
Die Bedeutung des automatischen Schließen erstreckt sich über zahlreiche Anwendungsbereiche in der KI, darunter das maschinelle Lernen, die Robotik, das automatische Beweisen in der Mathematik und die Entwicklung intelligenter Assistenzsysteme und viele weitere.
Durch die Verbesserung der Schlussfolgerungsfähigkeiten können Systeme entwickelt werden, die komplexe Probleme besser lösen, was zu intelligenteren und autonomeren Technologien führt. Bei der Entwicklung von Heuristiken im Bereich des automatischen Schlussfolgerns, die auf neuronalen Netzen basieren, besteht erhebliches Forschungspotenzial.
In Rahmen dieser Arbeit wurde eine Heuristik entwickelt, bei der ein Sprachmodell speziell darauf trainiert wurde, die Passgenauigkeit von logischen Symbolen in Bezug auf Beweisaufgaben zu beurteilen. Dieses Training basiert auf einer Analyse von Beweisen und deren Beweisprozesse, die darauf abzielt, vorherzusagen, wie gut ein bestimmtes Symbol zu einer zu beweisenden Aufgabe passt.
Das Hauptziel war es, den Beweisprozess durch gezielte Vorschläge des neuronalen Netzes zu verbessern, indem es die relevantesten Symbole für den Beweiser identifiziert und hervorhebt.
Die durchschnittliche Anzahl der Klauseln pro Beweisprozess konnte so reduziert werden, und es wurden neue, zuvor unentdeckte Beweise gefunden. Allerdings nahm die Gesamtzahl der gefundenen Beweise ab.
Obwohl bereits Fortschritte gemacht wurden, ist weiterhin umfangreiche Forschung erforderlich, und einige Fragestellungen bleiben im Rahmen dieser Arbeit ungeklärt. Diese lassen sich in drei Hauptkategorien unterteilen: Herausforderungen bei der Erzeugung der Trainingsdaten, Fragen im Bereich des Sprachmodells sowie Unsicherheiten bezüglich der Heuristik selbst.
Die Herausforderungen bei der Erzeugung der Trainingsdaten sind vielschichtig und umfassen mehrere Aspekte:

1. Die Bedeutung der Symbole und deren Übersetzung bedürfen einer gründlichen Prüfung, um sicherzustellen, dass sie ausreichend genau sind. Möglicherweise

müssen bessere Methoden für die Verbalisierung entwickelt werden, um die Trainingsdaten zu verbessern.
2. Es besteht die Möglichkeit, dass die Berechnung der Trainingsdaten mithilfe von TF-IDF-Werten und der Häufigkeit nicht optimal war. Alternativen müssen möglicherweise in Betracht gezogen werden, um die Qualität der Trainingsdaten zu erhöhen.
3. Ein weiteres Problem liegt darin, dass bei der Generierung von Trainingsdaten aus Beweisen häufig eine erhebliche Ungleichheit zwischen positiven und negativen Trainingsdaten besteht, was zu einer extremen Ungleichgewichtung führen kann. In der Arbeit wurden einige Ansätze zur Bewältigung dieses Problems diskutiert und vorgestellt, jedoch könnten möglicherweise alternative und verbesserte Methoden existieren.

Diese Herausforderungen zeigen, dass die Erzeugung hochwertiger Trainingsdaten ein komplexes Unterfangen ist und weitere Forschung erforderlich ist, um effektive Lösungen zu entwickeln.
Die Herausforderungen bei der Auswahl des richtigen Sprachmodells und des Trainings umfassen ebenfalls mehrere Aspekte:

1. Eventuell könnte ein anderes Sprachmodell bessere Ergebnisse liefern als das in dieser Arbeit verwendete.
2. Die Auswahl der Verlustfunktion stellt eine bedeutende Variable dar, auch wenn in dieser Arbeit bereits mehrere Varianten ausprobiert wurden.
3. Es bleibt die Frage, welche Batch-Größe optimal ist, um das Training effizient und effektiv zu gestalten.
4. Die Festlegung des idealen Zeitpunkts, um das Training optimalerweise zu stoppen, um Überanpassung zu vermeiden und die Generalisierungsfähigkeit des Modells zu maximieren.

Das sind jedoch eher allgemeine Fragestellungen, die sich immer wieder beim Training von Sprachmodellen ergeben.
Es ergeben sich ebenfalls viele Unsicherheiten bezüglich der in dieser Arbeit entwickelten Heuristik.

1. Ist der Vergleich in Bezug auf die Bedeutung von Symbolnamen überhaupt sinnvoll? Hätte man stattdessen die ganze Klausel als Embedding wählen sollen? Dies hätte allerdings den Nachteil, dass in die Implementierung des Eprovers eingegriffen werden muss.
2. Es ist fraglich, ob der Vergleich der Embedding-Vektoren über die Kosinus-Ähnlichkeit nicht besser durch einen anderen Vergleich erfolgen sollte.
3. War die Anzahl der Beweise für den Vergleich ausreichend, um eine aussagekräftige Bewertung der Herangehensweise der Heuristik zu ermöglichen?
4. Welche Metriken beim Beweisprozess sind für den Vergleich relevant?
5. Wie viele Symbole, die gut zu einem Beweisziel passen, und wie viele Symbole, die schlecht zu einem Beweisziel passen, sind optimal, um die besten Ergebnisse mit der Heuristik zu erzielen?

6. Inwieweit kann die in dieser Arbeit entwickelte Heuristik mit anderen Heuristiken kombiniert werden, und wie beeinflusst dies die Ergebnisse im Beweisprozess?

Um die in dieser Arbeit entwickelte Heuristik zuverlässig einzusetzen, bedarf es weiterführender Forschung, um die bestehenden Unsicherheiten zu klären.
Ein weiterer interessanter Ansatz für zukünftige Forschungen könnte die Untersuchung der Lernmöglichkeiten aus Beweisen während deren Führung sein.
Eine mögliche Methode, in Anlehnung an die in dieser Arbeit entwickelte Heuristik, könnte wie folgt aussehen:

1. **Für jeden erfolgreich gefundenen Beweis:**
   - **Positive Trainingsdaten:** Speichern der Symbole und ihrer Häufigkeiten, die im gefundenen Beweis vorkommen.
   - **Negative Trainingsdaten:** Erfassung der Häufigkeit von Symbolen, die während der Beweisführung vorkommen, jedoch nicht im gefundenen Beweis enthalten sind.
   - **Neutrale Trainingsdaten:** Bewertung mit Null für Symbole, die in vorherigen Beweisprozessen verwendet wurden, jedoch in diesem Beweisprozess nicht vorkommen.
2. **Nach einer bestimmten Anzahl von gefundenen Beweisen:**
   - **Generierung der Trainingsdaten:** Die Trainingsdaten werden nach einer gewissen Anzahl von gefundenen Beweisen wie folgt generiert, und das Sprachmodell wird mit diesen trainiert:
     - **Positive Trainingsdaten:** Die TF-IDF-Werte können einfach ermittelt werden, da sowohl die Symbole als auch deren Häufigkeiten für alle gefundenen Beweisen gespeichert wurden.
     - **Negative Trainingsdaten:** Diese sind bereits erstellt und basieren auf den gespeicherten Häufigkeiten der Symbole, die im Beweisprozess verwendet, jedoch nicht im gefundenen Beweis enthalten sind.
     - **Neutrale Trainingsdaten:** Auch diese sind bereits erstellt, indem Symbole gespeichert wurden, die in früheren Beweisprozessen vorkamen, aber im aktuellen Beweisprozess nicht verwendet wurden.

Im Vergleich zu dem Ansatz in dieser Bachelorarbeit, ein Sprachmodell vorzutrainieren und anschließend die Beweise mit Hilfe dieses Sprachmodells zu führen, bietet diese Methode einen strukturierteren Ansatz, um während des Führens von Beweisen zu lernen.
Zusammenfassend lässt sich sagen, dass diese Arbeit erste Schritte zur Verbesserung der Reasoning-Fähigkeiten von KI-Systemen durch die Entwicklung einer neuen Heuristik gemacht hat. Die beschriebenen Herausforderungen und offenen Fragen bieten eine solide Grundlage für zukünftige Forschungsarbeiten in diesem spannenden und bedeutenden Bereich der Informatik.

Literaturverzeichnis

ada. *Dokumentation AdamW.* https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html. Abgerufen am 18.04.2024.

ÁGDR18. Álvez, Javier, Itziar Gonzalez-Dios und German Rigau: *Applying the Closed World Assumption to SUMO-Based FOL Ontologies for Effective Commonsense Reasoning.* In: *European Conference on Artificial Intelligence*, 2018.

all. *Hugging Face Seite des all-MiniLM-L6-v2 Modells.* https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Abgerufen am 04.04.2024.

ALR14. Alvez, Javier, Paqui Lucio und German Rigau: *Adimen-SUMO: Reengineering an Ontology for First-Order Reasoning.* International Journal on Semantic Web and Information Systems, 8:80–116, 10 2014.

Ama23. Amaratunga, Thimira: *Understanding Large Language Models: Learning Their Underlying Concepts and Technologies.* Apress Berkeley, CA, 2023.

BG21. Bejani, Mahdi und Mehdi Ghatee: *A systematic review on overfitting control in shallow and deep neural networks.* Artificial Intelligence Review, 54:6391–6438, 03 2021.

BGMIM20. Babić, Karlo, Francesco Guerra, Sanda Martincic-Ipsic und Ana Meštrović: *A Comparison of Approaches for Measuring the Semantic Similarity of Short Texts Based on Word Embeddings.* Journal of information and organizational sciences, 44:231–246, 12 2020.

BKKS96. Bibel, W., D. Korn, C. Kreitz und S. Schmitt: *Problem-oriented applications of automated theorem proving.* In: Calmet, Jacques und Carla Limongelli (Herausgeber): *Design and Implementation of Symbolic Computation Systems*, Seiten 1–21, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

Cal03. Callan, Robert: *Neuronale Netze im Klartext.* Pearson Studium, 2003.

dat. *PyTorch Tutorial Datasets & DataLoaders.* https://pytorch.org/tutorials/beginner/basics/data_tutorial.html. Abgerufen am 18.04.2024.

Dav89. Davis, Ruth E: *Truth, deduction, and computation : logic and semantics for computer science*. Spektrum, 1989.

epr. *Eprover Awards*. `https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/Awards.html`. Abgerufen am 04.04.2024.

FM23. Filip, Bartek und Suda Martin: *How Much Should This Symbol Weigh? A GNN-Advised Clause Selection*. In: *Proceedings of 24th International Conference on Logic*, Band 94, Seiten 96–111, 2023.

GOS09. Guarino, Nicola, Daniel Oberle und Steffen Staab: *What Is an Ontology?*, Seiten 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

hub. *HuberLoss*. `https://pytorch.org/docs/stable/generated/torch.nn.HuberLoss.html`. Abgerufen am 09.05.2024.

JM23. Jurafsky, Daniel und James H. Martin: *Speech and Language Processing (3rd ed. draft)*, Kapitel Vector Semantics and Embeddings, Seiten 1–34. unpublished, draft available under `https://web.stanford.edu/~jurafsky/slp3/6.pdf`, 2023. Abgerufen am 30.05.2024.

loa. *Load pretrained instances with an AutoClass*. `https://huggingface.co/docs/transformers/autoclass_tutorial`. Abgerufen am 18.04.2024.

McK21. McKeown, John: *Clause representation for proof guidance using neural networks*. Master-Abschlussarbeit, University of Miami, 2021.

Mel96. Melvin, Fitting: *First-order logic and automated theorem proving*. Springer, New York, 2. Auflage, 1996.

MH07. Mpagouli, Aikaterini und Ioannis Hatzilygeroudis: *Converting first order logic into natural language: A first level approach*. In: *Current Trends in Informatics: 11th Panhellenic Conference on Informatics, PCI*, Seiten 517–526, 2007.

mod. *PyTorch Saving and Loading Models*. `https://pytorch.org/tutorials/beginner/saving_loading_models.html`. Abgerufen am 18.04.2024.

mse. *Dokumentation MSELoss*. `https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html`. Abgerufen am 18.04.2024.

NP01. Niles, Ian und Adam Pease: *Towards a standard upper ontology*. In: *2nd International Conference on Formal Ontology in Information Systems, FOIS 2001, Ogunquit, Maine, USA, October 17-19, 2001, Proceedings*, Seiten 2–9. ACM, 2001.

NSS20. Nahku Saidy, Hanna Siegfried, Stephan Schulz und Geoff Sutcliffe: *Cutting down the TPTP language (and others)*. In: *PAAR+SC-Square 2020 Practical Aspects of Automated Reasoning and Satisfiability Checking and Symbolic Computation Workshop 2020*, 2020.

PC20. Pilehvar, Mohammad Taher und José Camacho-Collados: *Embeddings in Natural Language Processing: Theory and Advances in*

Vector Representations of Meaning. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2020.

pyt. *PyTorch documentation*. `https://pytorch.org/docs/stable/index.html`. Abgerufen am 12.05.2024.

Rei09. Reimer, Kerstin: *Bootstrapping und andere Resampling-Methoden*, Seiten 521–536. Gabler Verlag, Wiesbaden, 2009.

RN12. Russell, Stuart und Peter Norvig: *Künstliche Intelligenz Ein moderner Ansatz*. Pearson, München, 3. Auflage, 2012.

Rob04. Robertson, Stephen: *Understanding Inverse Document Frequency: On Theoretical Arguments for IDF*. Journal of Documentation - J DOC, 60:503–520, 10 2004.

ŘS10. Řehůřek, Radim und Petr Sojka: *Software Framework for Topic Modelling with Large Corpora*. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, Seiten 45–50, Valletta, Malta, Mai 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

RW23. Robeson, Scott M. und Cort J. Willmott: *Decomposition of the mean absolute error (MAE) into systematic and unsystematic components*. PLOS ONE, 18(2):1–8, 02 2023.

Sch00a. Schulz, Stephan: *Learning Search Control Knowledge for Equational Deduction*. Dissertation-Abschlussarbeit, Technischen Universität München, 2000.

Sch00b. Schöning, Uwe: *Logik für Informatiker*. Spektrum, Berlin, 5. Auflage, 2000.

Sch02. Schulz, Stephan: *E - A Brainiac Theorem Prover*. AI Communications, 15, 09 2002.

Sch21. Schulz, Stephan: *E 2.6 User Manual*, 2021.

Sch22. Schon, Claudia: *Selection Strategies for Commonsense Knowledge*. CoRR, abs/2202.09163, 2022.

SM16. Schulz, Stephan und Martin Möhrmann: *Performance of Clause Selection Heuristics for Saturation-Based Theorem Proving*. In: Olivetti, Nicola und Ashish Tiwari (Herausgeber): *Automated Reasoning*, Seiten 330–345, Cham, 2016. Springer International Publishing.

SS67. Schwartz, J.T. und A.M. Society: *Mathematical Aspects of Computer Science*, Kapitel A REVIEW OF AUTOMATIC THEOREM-PROVING, Seiten 1–18. American Mathematical Society. Proceedings of Symposia in Applied Mathematics. American Mathematical Society, 1967.

SS98. Sutcliffe, Geoff und Christian Suttner: *The TPTP Problem Library*. Journal of Automated Reasoning, 21:177–203,, 10 1998.

SUP. *Präsentation: Superposition*. `https://www.eprover.org/EVENTS/Superposition-25/TutorialSP_ho.pdf`. Abgerufen am 28.02.2024, Tutorial für The 25th Conference on Automated Deduction.

SZK06. Shalabi, Luai, Shaaban Zyad und Basil Kasasbeh: *Data Mining: A Preprocessing Engine*. Journal of Computer Science, 2:735–739, 09 2006.

TBRP15. Torgo, Luís, Paula Branco, Rita P. Ribeiro und Bernhard Pfahringer: *Resampling strategies for regression*. Expert Systems, 32(3):465–476, 2015.

tpt. *Spezifikation der TPTP Syntax*. `https://tptp.org/TPTP/SyntaxBNF.html`. Abgerufen am 03.04.2024.

tra. *PyTorch Tutorial Training with PyTorch*. `https://pytorch.org/tutorials/beginner/introyt/trainingyt.html`. Abgerufen am 18.04.2024.

A

Selbstständigkeitserklärung

☒ Diese Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

☐ Diese Arbeit wurde als Gruppenarbeit angefertigt. Meinen Anteil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser:

Meine eigene Leistung ist:

31.05.2024
Datum

Fabian Koßmann
Unterschrift der Kandidatin/des Kandidaten